

“

INTRODUCCIÓN A NODE JS

Es un **entorno de ejecución**
que nos permite ejecutar
Javascript por **fuera** de un
navegador.



ARQUITECTURA NODE JS

Todos los navegadores presentan un **motor de Javascript** para leer y renderizar código JS. Esto hace que el lenguaje dependa sí o sí de un navegador para poder ejecutarse.

Los navegadores entre sí utilizan distintos motores, y es por esta variedad que a veces un mismo código JS puede comportarse de manera diferente dependiendo del navegador en el que se esté ejecutando.



Motor CHAKRA



Motor SpiderMonkey



Motor v8

3

ARQUITECTURA NODE JS

Node JS está construido bajo el motor **v8** de Google Chrome. Esto lo convierte en un entorno de ejecución para Javascript y logra que el lenguaje deje de depender del navegador para poder ejecutarse.

De esta forma, podemos programar tanto el Front-end como el Back-end en un mismo lenguaje: **Javascript**.



INSTALANDO NODE JS

Lo primero que hay que hacer es descargar Node.js desde su página oficial: nodejs.org/es

Conjuntamente con Node.js se va a instalar el gestor de paquetes **NPM**, que veremos a fondo más adelante.

Para verificar que se instaló correctamente, abrir una terminal y ejecutar el comando `node -v` ó `node --version`.



Tener en cuenta que al instalar Node no estamos instalando un software si no un entorno de ejecución.

5

PROBANDO NODE JS

Para testear NodeJS, crear una carpeta llamada **Node**.

Abrir el editor de texto Visual Studio Code. Ir a Archivo/Abrir carpeta y seleccionar la carpeta que creamos recién.

Crear un archivo llamado **prueba.js** y escribir el siguiente script:

```
console.log('Probando Node!');
```

Abrir una terminal. Para eso, ir a Terminal/Nueva terminal, o ejecutar el atajo **ctrl + shift + n**.

En la terminal escribir el siguiente comando:

```
node prueba.js
```

Si todo anduvo bien, veremos en la terminal el mensaje:
Probando Node!

6

“

NODE PACKAGE MANAGER **NPM**

NPM es el **gestor de paquetes** de Node y nos permite descargar e instalar **librerías** para **incorporar** a nuestro proyecto.



INTRODUCCIÓN A NPM

Cuando instalamos Node en nuestras computadoras, se instalan múltiples librerías para poder usar **globalmente**, y, conjuntamente, se instala **NPM**, el gestor de paquetes de Node.

A través de él vamos a poder instalar las librerías que consideremos necesarias para el desarrollo de nuestra aplicación. Podemos instalarlas **localmente**, disponibles para usar en un proyecto en específico, ó **globalmente**, disponibles para usar cada vez que queramos.



3

QUÉ SON LAS LIBRERÍAS

Son **bloques de código** que nos permiten abordar soluciones específicas dentro de la aplicación que estemos desarrollando. En un entorno de desarrollo web, hay situaciones que se repiten una y otra vez. Las librerías llegan para **facilitar** esas problemáticas que sabemos que nos vamos a cruzar mientras desarrollamos nuestra aplicación. Manejar la subida de archivos, validar un formulario, o restringir el acceso a un usuario que no está registrado son algunas de ellas.



USANDO NPM

Cuando se instala Node, se genera un comando `npm` para usar en la terminal.

Lo primero que hay que hacer para usar npm es **inicializar** nuestro proyecto Node usando el comando `npm init`. Este comando creará un archivo **package.json**, dentro del cual se irán guardando todas las configuraciones del proyecto. Por el momento, la característica que más nos interesa de este archivo es la propiedad `"main"`. La misma hace referencia al **entry point**, es decir, el punto de entrada a nuestra aplicación, en donde pondremos el nombre de nuestro archivo principal, que, por convención solemos llamar `app.js`.

package.json

```
{
  "name": "app",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Dirección al **Entry Point** (punto de entrada)

5

6

INSTALANDO LIBRERÍAS

Para instalar una librería usamos el siguiente comando:

```
npm install PACKAGE --save
```

 en donde reemplazaremos la palabra `PACKAGE` por el nombre de la librería que queremos instalar.

El comando `--save` guarda dentro del **package.json**, en la propiedad `"dependencies"`, una referencia a la librería que estamos instalando.

package.json

```
"main": "app.js",
"scripts": {
  "test": "echo \\"Error: no test specified\\\" && exit 1"
},
"author": "",
"license": "ISC",
"dependencies": {
  "moment": "^2.24.0"
}
```

Referencia a la/las librerías que instalaremos en nuestro proyecto.

7

8

Dentro de la carpeta **Node**

Modules se irán creando las carpetas de las librerías que instalaremos.

Cada una contendrá los **archivos necesarios** para poder trabajar con esa librería dentro del proyecto.



NODE MODULES

Son **estructuras de código** que, en conjunto, conforman la **totalidad** de nuestra aplicación y configuran su usabilidad.



QUÉ ES UN MÓDULO

Un módulo es un bloque de **código reusable**, una unidad funcional, cuya existencia o no, no altera el comportamiento de otros bloques de código.

A partir de eso, Node propone atomizar nuestro código, es decir, **fragmentarlo** en pequeños módulos, en donde cada uno tendrá una funcionalidad específica para alcanzar un objetivo definido.

Existen tres tipos de módulos:

Los **módulos nativos**, aquellos que ya vienen instalados.

Los **módulos de terceros**, aquellos que podemos instalar usando NPM.

Los **módulos creados**, aquellos que definimos nosotros.



3

CÓMO REQUERIR UN MÓDULO

Para requerir un módulo, sin importar de qué tipo sea, hace falta situarse dentro del archivo en el que queremos incorporarlo y hacer uso de la función nativa de node `require()`. La misma recibe como parámetro un string, que será el **nombre** del módulo.

Esta función devuelve un **objeto literal**, por lo tanto es importante guardar la ejecución en una **variable**, para poder acceder, a través del **dot notation**, a todas las propiedades y funcionalidades del módulo.

```
let modulo = require('nombreModulo');
modulo.propiedad;
modulo.funcionalidad();
```

Por **convención**, el **nombre** de la variable que almacene el módulo que estamos requiriendo, suele recibir el **mismo** nombre del módulo, o una **abreviatura**.



5

MÓDULO NATIVO

Para requerir un módulo nativo usamos la función `require()` y le pasamos como argumento el nombre del módulo que queremos requerir.

En [este link](#) vas a encontrar los módulos que vienen incluidos cuando instalamos Node, listados en orden alfabético a la izquierda.

```
{ const fs = require('fs');
```

MÓDULO DE TERCEROS

Para requerir un módulo de terceros, primero hay que instalarlo usando el comando `npm install PACKAGE --save`.

Una vez instalado, usamos la función `require()` y le pasamos como argumento el nombre del módulo que instalamos.

```
>_ npm install moment --save
```

```
{ const moment = require('moment');
```

7

8

MÓDULO CREADO

Para requerir un módulo creado por nosotros, primero hay que crear un archivo con extensión `.js` y dentro del mismo escribir el script que necesitemos.

Una vez definido nuestro código, tenemos que dejarlo accesible para poder importarlo dentro de nuestra aplicación. Para eso hay que hacer uso del **objeto nativo** `module` y de su propiedad `exports`. Al mismo le asignaremos el nombre de la variable que contenga la información que queremos exponer.

{ código }

```
const series = [
  {titulo: 'Mad Men', temporadas: 7},
  {titulo: 'Breaking Bad', temporadas: 5},
  {titulo: 'Seinfeld', temporadas: 9},
];
module.exports = series;
```

Definimos un **array** de series, en donde en cada posición hay un objeto literal con las propiedades `título` y `temporadas`, contenido la información de cada serie.

9

10

{ código }

```
const series = [
  {titulo: 'Mad Men', temporadas: 7},
  {titulo: 'Breaking Bad', temporadas: 5},
  {titulo: 'Seinfeld', temporadas: 9},
];
module.exports = series;
```

Hacemos uso del objeto `module` y su propiedad `exports`, y le asignamos la variable que queremos exportar, en este caso, `series`.

Tener en cuenta que siempre que queramos exportar módulos de un script, tendremos que escribir esta línea al final del mismo.

MÓDULO CREADO

Una vez que exportamos nuestro módulo, vamos al archivo en donde lo queremos importar y usamos la función `require()`.

En este caso le pasamos como argumento la **ruta** hacia el script donde se encuentra el módulo que queremos requerir. Para eso, usamos el `..`. De e `./` forma le estamos indicando a Node que el camino para llegar a ese módulo empieza **desde** donde estamos parados (`app.js`) **hasta** el **nombre** del archivo que le pasemos.



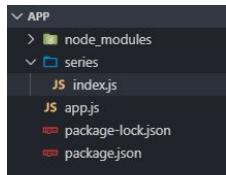
Como buena práctica, se suelen almacenar los módulos que creamos dentro de una carpeta con el mismo nombre del módulo que estamos por crear.

11

12

MÓDULO CREADO

Cuando nos referimos a archivos de Javascript no hace falta escribir la extensión.



```
{ } const series = require('./series/index');
```

Para poder ver **todo** lo que trae consigo un módulo, podemos hacer un `console.log()` de la **variable** en la que lo almacenamos.



“

Las **variables**, son **espacios de memoria** en la computadora donde podemos **almacenar** distintos tipos de **datos**.



LAS VARIABLES

TIPOS DE VARIABLE

En Javascript existen tres tipos de variables:

- var
- let
- const

Para declarar una variable escribimos el tipo y el nombre que le queremos dar a la variable:

```
var nombre;
let contador;
const URL;
```

Veamos cada parte en más detalle...

DECLARACIÓN DE UNA VARIABLE

`var nombreSignificativo;`

`var`

La palabra reservada `var` le indica a Javascript que vamos a **declarar una variable de tipo var**.

`Nombre`

Sólo puede estar formado por letras, números y los símbolos \$ y _ (guion bajo). No pueden empezar con un número. No deberían contener ñ o caracteres con acentos.



Es una **buenas prácticas** que los nombres de las variables usen el formato `camelCase`, como `variableEjemplo` en vez de `variableejemplo` o `variable_ejemplo`.

DECLARACIÓN DE UNA VARIABLE

“

```
var miVariable;
```

i...no es lo mismo que...!

```
var MiVariable;
```

Las **buenas prácticas**, si bien no son obligatorias para que nuestro **código** funcione, van a permitir que el mismo sea **más fácil de leer y de mantener**.



⚠️ Javascript es un lenguaje que **hace diferencia entre MAYÚSCULAS y minúsculas**. Por eso es bueno seguir un estándar a la hora de escribir nombres.

ASIGNACIÓN DE UN VALOR

Cuando declaramos una variable, también podemos al mismo tiempo asignarle un valor. Eso lo hacemos con el operador de asignación.

```
var miApodo = 'Hackerman';
```

↓ ↓ ↓

Nombre **Asignación** **Valor**

El nombre que nos va a servir para identificar nuestra variable cuando necesitemos usarla.

Le indica a JavaScript que queremos guardar el valor de la derecha en la variable de la izquierda.

Lo que vamos a guardar en nuestra variable. En este caso, un texto.

ASIGNACIÓN DE UN **VALOR**

La primera vez que declaramos una variable, es necesaria la palabra reservada **var**.

```
var miApodo = 'Hackerman';
```

Una vez que la variable ya fue declarada, le asignamos valores sin **var**.

```
miApodo = 'El Barto';
```

Nuestra variable `guardará siempre el último valor asignado`, eso quiere decir que si volvemos a asignarle un valor, pisamos el anterior.

DECLARACIÓN CON LET

Estas variables se declaran de una manera similar con la diferencia que utilizamos la **palabra reservada let**.

```
let contador = 0;
```

La principal diferencia entre `var` y `let` es que `let` sólo será accesible en el bloque de código en el que fue declarada.

Los bloques de código son normalmente determinados por las llaves {}.

Veamos un ejemplo:

VAR

```
if (true) {  
    var nombre = "Juan";  
}
```

```
console.log(nombre);
// Ok, muestra "Juan"
```

Cuando usamos **var** JavaScript ignora los bloques de código y convierte nuestra variable en global.

Eso quiere decir que si hay otra variable **nombre** en nuestro código, seguramente estemos pisando su valor.

LET

```
if (true) {  
  let nombre = "Juan";  
}  
  
console.log(nombre)
```

Cuando usamos `let` JavaScript respeta los bloques de código. Eso quiere decir que `nombre` no podrá ser accedida fuera del `if`.

También quiere decir que podemos tener variables con el mismo nombre en diferentes bloques de nuestro código.

DECLARACIÓN CON CONST

Las variables **const** se declaran con la palabra reservada **const**

```
const EMAIL = "mi.email@hotmail.com";
```

Las variables declaradas con **const** funcionan igual que las variables **let**, estarán disponibles sólo en el bloque de

Al contrario que `let`, una vez que les asignemos un valor

```
EMAIL = "mi.nuevo.email@gmail.com";
// Error de asignación, no se puede cambiar el
valor de un const
```

DECLARACIÓN CON CONST

Las variables **const** se declaran con la palabra reservada **const**.

`const EMAIL = "mi.email@hotmail.com";`

Las variables declaradas con la palabra reservada `const` suele declararse en MAYÚSCULAS, iésto es una buena práctica.

Al contrario que **let**, una vez que les asignemos un valor, no podremos cambiarlo.

```
EMAIL = "mi.nuevo.email@gmail.com";
// Error de asignación, no se puede cambiar el
valor de un const.
```

DECLARACIÓN CON LET O CONST

Como dijimos antes tanto **let** como **const** son accesibles dentro del bloque donde son declaradas.

Por esta razón sólo podemos declararlas una vez, si volvemos a declararlas, JavaScript nos devolverá un error.

```
let contador = 0;  
let contador = 1;  
// Error de re-declaración de la variable  
  
const EMAIL = "mi.email@hotmail.com";  
const EMAIL = "mi.nuevo.email@hotmail.com";  
// Error de re-declaración de la variable
```

“

Las **palabras reservadas** como **var**, **let** y **const** sólo pueden utilizarse para el propósito que fueron creadas.

No pueden ser utilizadas como: **nombre de variables, funciones, métodos o identificadores de objetos.**



LOS TIPOS DE DATOS

Los **tipos de datos** le permiten a JavaScript **conocer las características y funcionalidades** que estarán disponibles para ese **dato**.



NUMÉRICOS (number)

```
0  
let edad = 35; // número entero  
let precio = 150.65; // con decimales
```

Como JavaScript está escrito en inglés usaremos un punto para separar los decimales.

CADERAS DE CARACTERES (string)

```
0  
let nombre = 'Mamá Luchetti'; // comillas simples  
let ocupacion = "Master of the sopas"; // comillas dobles tienen el mismo resultado
```

LÓGICOS O BOOLEANOS (boolean)

```
0  
let laCharlaEstaReCopada = true;  
let hayAsadoAlFinal = false;
```

OBJETO (object)

A diferencia de otros tipos de datos que pueden contener un solo dato, los objetos son **colecciones** de datos y en su interior pueden existir todos los anteriores.

Los podemos reconocer porque se declaran con llaves {}.

```
let persona = {  
    nombre: 'Javier', // string  
    edad: 34, // number  
    soltero: true // boolean  
}
```

ARRAY

Al igual que los objetos, los arrays son colecciones de datos. Los podemos reconocer porque se declaran con corchetes [].

Los arrays son un tipo especial de objetos, por eso **no los consideramos como un tipo de dato más**.

Los mencionamos de manera especial porque son muy comunes en todo tipo de código.

```
0  
let comidasFavoritas = ['Milanesa napolitana',  
'Ravióles con bolognesa', 'Pizza calabresa'];  
  
let numerosSorteados = [12, 45, 56, 324, 452];
```

3

LOS TIPOS DE DATOS ESPECIALES

5

“

Los tipos de datos especiales

le permiten a JavaScript determinar **estados especiales** que pueden tener los **datos**.



NaN (NOT A NUMBER)

```
let malaDivision = "35" / 2; // NaN no es un número
```

NULL (VALOR NULO)

Lo asignamos nosotros para indicar un valor vacío o desconocido.

```
let temperatura = null; // No llegó un dato, algo falló
```

UNDEFINED (valor sin definir)

Las variables tienen un valor indefinido hasta que les asignamos un valor.

```
let otraVariable; // undefined, no tiene valor  
otraVariable = "¡Hola!"; // Ahora si tiene un valor
```

Los comentarios son partes de nuestro código que **no se ejecutan**.

Siempre comienzan con dos barras inclinadas **//**

Los usamos para explicar lo que estamos haciendo y **dejar información útil** para nuestro equipo o para nuestro yo del futuro.



```
// Math.round() retorna el valor redondeado al entero  
más cercano.  
let redondeado = Math.round(20.49);
```

“

Los **operadores** nos permiten **manipular el valor** de las variables, realizar **operaciones** y **comparar** sus valores



LOS OPERADORES

DE ASIGNACIÓN

Asignan el valor de la derecha en la variable de la izquierda.

```
0 let edad = 35; // Asigno el número 35 a edad
```

ARITMÉTICOS

Nos permiten hacer operaciones matemáticas, devuelven el resultado de la operación.

```
0 10 + 15 // Suma → 25  
10 - 15 // Resta → -5  
10 * 15 // Multiplicación → 150  
15 / 10 // División → 1.5
```

ARITMÉTICOS (CONTINUACIÓN)

```
0 15++ // Incremento, es igual a 15 + 1 → 16  
15-- // Decremento, es igual a 15 - 1 → 14
```

```
0 15 % 5 // Módulo, el resto de dividir 15 entre 5 → 0  
15 % 2 // Módulo, el resto de dividir 15 entre 2 → 1
```

El operador de módulo (%) nos devuelve el resto de una división.

$$\begin{array}{r} 15 \\ \hline 5 \\ \hline 0 \end{array} \quad \begin{array}{r} 15 \\ \hline 2 \\ \hline 1 \end{array}$$

3

4

“

Los **operadores** aritméticos siempre **devolverán el resultado numérico** de la **operación** que se esté realizando.



DE COMPARACIÓN SIMPLE

Comparan dos valores, devuelven verdadero o falso.

```
0 10 == 15 // Igualdad → false  
10 != 15 // Desigualdad → true
```

DE COMPARACIÓN ESTRICTA

Comparan el valor y el tipo de dato también.

```
0 10 === "10" // Igualdad estricta → false  
10 !== 15 // Desigualdad estricta → true
```

En el primer caso el valor es 10 en ambos casos, pero los tipos de datos son number y string. Como estamos comparando que ambos (valor y tipo de dato) sean iguales, el resultado es **false**.

6

DE COMPARACIÓN (CONTINUACIÓN)

Comparan dos valores, devuelven verdadero o falso.

```
{ } 15 > 15 // Mayor que → false  
15 >= 15 // Mayor o igual que → true  
10 < 15 // Menor que → true  
10 <= 15 // Menor o igual que → true
```

 Siempre debemos escribir el símbolo mayor (>) o menor (<) antes que el igual (>= o <=). Si lo hacemos al revés (= o ==) JavaScript lee primero el operador de asignación = y luego no sabe qué hacer con el mayor (>) o el menor (<).

“

Los **operadores de comparación** siempre **devolverán** un booleano, es decir **true** o **false**, como resultado.



7

LÓGICOS

Permiten combinar valores booleanos, el resultado también devuelve un booleano.

Existen tres operadores **y** (and), **o** (or), **negación** (not).

AND (&&) todos los valores deben evaluar como **true**.

```
{ } (10 > 15) && (10 != 20) // false  
FALSE → TRUE → FALSE
```

```
{ } (12 % 4 == 0) && (12 != 24) // true  
TRUE → TRUE → TRUE
```

LÓGICOS (CONTINUACIÓN)

OR (||) al menos un valor debe evaluar como **true**.

```
{ } (10 > 15) || (10 != 20) // true  
FALSE → TRUE → TRUE
```

```
{ } (12 % 4 == 0) && (12 != 24) // true  
TRUE → TRUE → TRUE
```

NOT (!) niega la condición, si era true, será false y viceversa

```
{ } !false // true  
!(10 > 15) // true
```

9

10

“

Los **operadores lógicos** siempre **devolverán** un booleano, es decir **true** o **false**, como resultado.



DE CONCATENACIÓN

Sirve para unir cadenas de texto. Devuelve otra cadena de texto.

```
{ } let nombre = 'Teodoro';  
let apellido = 'García';  
let nombreCompleto = nombre + ' ' + apellido;
```

Si mezclamos otros tipos de datos, estos se convierten a cadenas de texto.

```
{ } let fila = 'M';  
let asiento = 7;  
let ubicacion = fila + asiento; // 'M7' como string
```

12

“

REPASO FUNCIONES

Una función es un **bloque de código** que podemos invocar todas las veces que necesitemos.

Puede realizar una **tarea específica** y **retornar** un valor.

Nos permite **agrupar el código** que vayamos a **usar muchas veces**.



1. DECLARACIÓN Y ESTRUCTURA

ESTRUCTURA BÁSICA

Palabra reservada

Usamos la palabra **function** para indicarle a Javascript que vamos a escribir una función.

```
function sumar (a,b) {
    return a + b;
}
```

ESTRUCTURA BÁSICA

Nombre

Definimos un **nombre** para referirnos a nuestra función al momento de querer invocarla.

```
function sumar(a,b) {
    return a + b;
}
```

ESTRUCTURA BÁSICA

Parámetros

Escribimos los paréntesis y dentro de ellos los parámetros de la función. Si lleva más de uno, los sepáramos usando comas ,.

Si la función no lleva parámetros, escribimos los paréntesis sin nada adentro ().

```
function sumar (a, b) {
    return a + b;
}
```

ESTRUCTURA BÁSICA

Parámetros

Dentro de nuestra función podremos acceder a los parámetros como si fueran variables. Es decir, con solo escribir los nombres de los parámetros, podremos trabajar con ellos.

```
function sumar(a, b) {  
    return a + b;  
}
```

ESTRUCTURA BÁSICA

Cuerpo

Entre las llaves de apertura y de cierre escribimos la lógica de nuestra función, es decir, el código que queremos que se ejecute cada vez que la invoquemos.

```
function sumar(a, b) {  
    return a + b;  
}
```

ESTRUCTURA BÁSICA

El retorno

Es muy común a la hora de escribir una función que queramos devolver al exterior el resultado del proceso que estamos haciendo dentro de ella.

Para eso utilizamos la palabra reservada **return** seguida de lo que queramos retornar.

```
function sumar(a, b) {  
    return a + b;  
}
```

FUNCIONES DECLARADAS

Son aquellas que se declaran usando la **estructura básica**. Reciben un **nombre formal** a través del cual la invocaremos.

```
function hacerHelado(cantidad) {  
    return '🍦'.repeat(cantidad)  
}
```



Se cargan **antes** de que cualquier código sea ejecutado.

FUNCIONES EXPRESADAS

Son aquellas que se **asignan como valor** a una variable. El nombre de la función será el **nombre de la variable** que declaremos.

```
let hacerSushi = function(cantidad) {  
    return '🍣'.repeat(cantidad)  
}
```



Se cargan cuando el intérprete alcanza la línea de código donde se encuentra la función.

2.

INVOCACIÓN

INVOCANDO UNA FUNCIÓN

La forma de **invocar** (ejecutar) una función es escribiendo su nombre seguido de apertura y cierre de paréntesis.

```
nombreFuncion();
```

En caso de querer ver o guardar el dato que **retorna**, será necesario almacenarlo en una variable, o hacer un `console.log` de la ejecución.

```
let resultado = nombreFuncion();
console.log(nombreFuncion());
```

INVOCANDO UNA FUNCIÓN

Si la función espera argumentos, se los podemos pasar dentro del paréntesis.

Es importante respetar el orden si hay más de un parámetro ya que JavaScript los asignará en el orden que lleguen.

```
function saludar(nombre, apellido) {
    return 'Hola ' + nombre + ' ' + apellido;
}
saludar('Robertito', 'Rodríguez');
// retorna 'Hola Robertito Rodríguez'
```

13

14

INVOCANDO UNA FUNCIÓN

También es importante tener en cuenta que cuando tenemos parámetros en nuestra función, Javascript va a esperar que se los pasemos como argumentos al ejecutarla.

```
function saludar(nombre, apellido) {
    return 'Hola ' + nombre + ' ' + apellido;
}
saludar(); // retorna 'Hola undefined undefined'
```



Al no haber recibido el argumento que necesitaba, Javascript le asigna el tipo de dato **undefined** a las variables nombre y apellido.

INVOCANDO UNA FUNCIÓN

Para este tipo de casos Javascript nos permite definir los **valores por defecto**.

Si agregamos un igual `=` luego del parámetro, podremos especificar su valor en caso de que no llegue ninguno.

```
function saludar(nombre = 'visitante',
    apellido = 'anónimo') {
    return 'Hola ' + nombre + ' ' + apellido;
}
saludar(); // retorna 'Hola visitante anónimo'
```

15

16

Los **parámetros** son las **variables** que escribimos cuando **definimos** la función.



Los **argumentos** son los **valores** que enviamos cuando **invocamos** la función.

3. SCOPE

El **scope** refiere al alcance que tiene una variable, es decir desde dónde podemos acceder a ella.

Los scopes **son definidos** principalmente **por las funciones**. Es fundamental dominarlo cuando trabajamos con ellas.



SCOPE LOCAL

En el momento en que declaramos una variable **dentro** de una función, la misma pasa a tener **alcance local**. Es decir, esa variable vive únicamente **dentro** de esa función.

Si quisieramos hacer uso de la variable por **fuera** de la función, no vamos a poder, dado que para **JavaScript** esa variable **no existe**.

```
function miFuncion() {
    // todo el código que escribamos dentro
    // de nuestra función, tiene scope local
}
```

{ código }

```
function hola() {
    let saludo = 'Hola ¿qué tal?';
    return saludo;
}

console.log(saludo);
```

Definimos la variable **saludo** dentro de la función **hola()**, por lo tanto su **scope es local**. Sólo dentro de esta función podemos acceder a ella.

{ código }

```
function hola() {
    let saludo = 'Hola ¿qué tal?';
    return saludo;
}

console.log(saludo);
```

Al querer hacer uso de la variable **saludo** por fuera de la función, Javascript no la encuentra y nos devuelve el siguiente error:

Uncaught ReferenceError: saludo is not defined

SCOPE GLOBAL

En el momento en que declaramos una variable **fueras** de cualquier función, la misma pasa a tener **alcance global**. Es decir, podemos hacer uso de ella desde cualquier lugar del código en el que nos encontremos, inclusive dentro de una función, y acceder a su valor.

```
// todo el código que escribamos fuera
// de las funciones, es global
let miVariable;
function miFuncion() {
    // Tenemos acceso a las variables globales
}
```

{ código }

```
let saludo = 'Hola ¿qué tal?';

function hola() {
    return saludo;
}
```

Declaramos la variable **saludo** por fuera de nuestra función, por lo tanto su **scope es global**.

Podemos hacer uso de ella desde cualquier lugar del código.

{ código }

```
let saludo = 'Hola ¿qué tal?';
```

```
function hola() {  
    return saludo;  
}
```

Dentro de la función `hola()` llamo a la variable `saludo`.

Su alcance es **global**, por lo tanto, Javascript sabe a qué variable me estoy refiriendo y ejecuta la función con éxito.

LOS CONDICIONALES IF / ELSE IF / ELSE

Nos permiten **evaluar condiciones** y realizar diferentes acciones **según el resultado** de esas evaluaciones.



CONDICIONAL SIMPLE

```
{ if (condición) {
    // código a ejecutar si la condición es verdadera
}
```

CONDICIONAL CON BLOQUE ELSE

```
{ if (condición) {
    // código a ejecutar si la condición es verdadera
} else {
    // código a ejecutar si la condición es falsa
}
```

CONDICIONAL CON BLOQUES ELSE IF

```
{ if (condición) {
    // código a ejecutar si la condición es verdadera
} else if (otra condición) {
    // código a ejecutar si la otra condición es verdadera
} else {
    // código a ejecutar si la condición es falsa
}
```

3

4

{ código }

```
let edad = 19;
let acceso = '';

if (edad < 16) {
    acceso = 'prohibido';
} else if (edad >= 16 && edad <= 18) {
    acceso = 'permitido sólo acompañado de un mayor';
} else {
    acceso = 'permitido';
}
```

{ código }

```
let edad = 19;
let acceso = '';

if (edad < 16) {
    acceso = 'prohibido';
} else if (edad >= 16 && edad <= 18) {
    acceso = 'permitido sólo acompañado de un mayor';
} else {
    acceso = 'permitido';
}
```

Declaramos la variable **edad** y le asignamos el número 19.

5

6

{ código }

```
let edad = 19;  
let acceso = '';  
  
if (edad < 16) {  
    acceso = 'prohibido';  
} else if (edad >= 16 && edad <= 18) {  
    acceso = 'permitido sólo acompañado de un  
mayor';  
} else {  
    acceso = 'permitido';  
}
```

Declaramos la variable **acceso** y le asignamos un string vacío, con la intención de asignarle un nuevo valor según el resultado que arrojen los condicionales declarados a continuación.

{ código }

```
let edad = 19;  
let acceso = '';  
  
if (edad < 16) {  
    acceso = 'prohibido';  
} else if (edad >= 16 && edad <= 18) {  
    acceso = 'permitido sólo acompañado de un  
mayor';  
} else {  
    acceso = 'permitido';  
}
```

Iniciamos el condicional. Nuestra primera condición evalúa si **edad** es menor a 16.

En caso de ser **verdadera**, le asignamos el string 'prohibido' a la variable **acceso**.

En este caso, la condición es **falsa**, por lo tanto Javascript pasa a evaluar la siguiente condición.

7

{ código }

```
let edad = 19;  
let acceso = '';  
  
if (edad < 16) {  
    acceso = 'prohibido';  
} else if (edad >= 16 && edad <= 18) {  
    acceso = 'permitido sólo acompañado de un  
mayor';  
} else {  
    acceso = 'permitido';  
}
```

Declaramos un bloque **else if** para contemplar una **segunda condición**:

Esta condición va a ser compuesta y va a requerir:

- que edad sea mayor o igual a 16
- que edad sea menor o igual a 18

La condición nuevamente es **falsa**, por lo tanto Javascript continúa leyendo el condicional.

{ código }

```
let edad = 19;  
let acceso = '';  
  
if (edad < 16) {  
    acceso = 'prohibido';  
} else if (edad >= 16 && edad <= 18) {  
    acceso = 'permitido sólo acompañado de un  
mayor';  
} else {  
    acceso = 'permitido';  
}
```

Como **ninguna** de las condiciones anteriores era **verdadera**, se ejecuta el código dentro del **else**.

Por lo tanto, ahora la variable **acceso** es igual al string 'permitido'.

9

“

Es una **buenas prácticas** inicializar las variables con el **tipo de dato** que van a almacenar.

```
>  
let texto = ''; // un texto vacío  
let numero = 0; // un número vacío
```



De esa manera queda más claro para qué se van a utilizar.

“



Los **arrays** nos permiten generar una **colección** de **datos ordenados**.

ARRAYS

ESTRUCTURA DE UN ARRAY

Utilizamos corchetes `[]` para indicar el **inicio** y el **fin** de un array. Utilizamos comas `,` para **separar** sus elementos.

Dentro, podemos almacenar la cantidad de elementos que queramos sin importar el tipo de dato de cada uno.

Es decir, podemos tener, en un mismo array datos de tipo string, number, boolean, y todos los demás.

```
{ } let miArray = ['Star Wars', true, 23];
```

POSICIONES DENTRO DE UN ARRAY

Cada dato de un array ocupa una posición numerada conocida como **índice**. La primera posición de un array **es siempre 0**.

```
let pelisFavoritas = ['Star Wars', 'Kill Bill', 'Alien'];  
          ↑   ↑   ↑  
          0   1   2
```

Para acceder a un elemento puntual de un array, nombramos al array y, **dentro** de los **corchetes**, escribiremos el **índice** al cual queremos acceder.

```
pelisFavoritas[2];  
// accedemos a la película Alien, el índice 2 del array
```

3

4

“

MÉTODOS DE ARRAYS I

Para JavaScript los arrays son un tipo especial de objetos.

Por esta razón disponemos de un montón de **métodos** muy útiles a la hora de trabajar con la información que hay adentro.



.push()

Agrega uno o varios **elementos al final** del array.

- Recibe uno o más elementos como parámetros
- Retorna la nueva longitud del array

```
var colores = ['Rojo', 'Naranja', 'Azul'];
colores.push('Violeta'); // retorna 4
console.log(colores);
// ['Rojo', 'Naranja', 'Azul', 'Violeta']
colores.push('Gris', 'Oro');
console.log(colores);
// ['Rojo', 'Naranja', 'Azul', 'Violeta', 'Gris', 'Oro']
```

.pop()

Elimina el **último elemento** de un array.

- No recibe parámetros
- Devuelve el elemento eliminado

```
var series = ['Mad Men', 'Breaking Bad', 'The Soprano'];
// creamos una variable para guardar lo que devuelve .pop()
var ultimaSerie = series.pop();
console.log(series); // ['Mad men', 'Breaking Bad']
console.log(ultimaSerie); // ['The Soprano']
```

3

4

.shift()

Elimina el **primer elemento** de un array.

- No recibe parámetros
- Devuelve el elemento eliminado

```
var nombres = ['Frida', 'Diego', 'Sofía'];
// creamos una variable para guardar lo que devuelve .shift()
var primerNombre = nombres.shift();

console.log(nombres); // ['Diego', 'Sofía']
console.log(primerNombre); // ['Frida']
```

.unshift()

Agrega uno o varios **elementos al principio** de un array.

- Recibe uno o más elementos como parámetros
- Retorna la nueva longitud del array

```
var marcas = ['Audi'];
marcas.unshift('Ford');
console.log(marcas); // ['Ford', 'Audi']

marcas.unshift('Ferrari', 'BMW');
console.log(marcas); // ['Ferrari', 'BMW', 'Ford', 'Audi']
```

5

6

.join()

Une los elementos de un array utilizando el separador que le especificaremos. Si no lo especificamos, utiliza comas.

- Recibe un separador (string), **opcional**.
- Retorna un string con los elementos unidos

```
var dias = ['Lunes','Martes','Miércoles','Jueves'];
var separadosPorComa = dias.join(',');
console.log(separadosPorComa);
// 'Lunes,Martes,Miércoles,Jueves'
var separadosPorGuion = dias.join(' - ');
console.log(separadosPorGuion);
// 'Lunes - Martes - Miércoles - Jueves'
```

7

.indexOf()

Busca en el array el elemento que recibe como parámetro.

- Recibe un elemento a buscar en el array
- Retorna el primer índice donde encontró lo que buscábamos. Si no lo encuentra, retorna un -1.

```
var frutas = ['Manzana','Pera','Frutilla'];
frutas.indexOf('Frutilla');
// Encontró lo que buscaba.
// Devuelve 2, el índice del elemento

frutas.indexOf('Banana');
// No encontró lo que buscaba. Devuelve -1
```

8

.lastIndexOf()

Similar a .indexOf(), con la salvedad de que empieza buscando el elemento por el final del array (de atrás hacia adelante).

En caso de haber elementos repetidos, devuelve la posición del primero que encuentre (osea el último si miramos desde el principio).

```
var clubes = ['Racing','Boca','Lanús','Boca'];
clubes.lastIndexOf('Boca');
// Encontró lo que buscaba. Devuelve 3

clubes.indexOf('River');
// No encontró lo que buscaba. Devuelve -1
```

9

.includes()

También similar a .indexOf(), con la salvedad que retorna un booleano.

- Recibe un elemento a buscar en el array
- Retorna true si encontró lo que buscábamos, false en caso contrario.

```
var frutas = ['Manzana','Pera','Frutilla'];
frutas.includes('Frutilla');
// Encontró lo que buscaba. Devuelve true

frutas.includes('Banana');
// No encontró lo que buscaba. Devuelve false
```

10

“

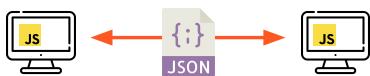
JSON JAVASCRIPT OBJECT NOTATION

Es un **formato de texto** sencillo utilizado para el **intercambio de datos** entre distintos **sistemas**.



LA POPULARIDAD DE JSON

En la web, la mayoría de las peticiones y sus respuestas viajan como texto plano, es decir, texto sin codificaciones especiales. JSON al ser una **cadena de texto simple**, es un **formato ideal para transmitir información** entre sitios y aplicaciones web. Especialmente si tenemos en cuenta que Javascript está presente en todos los navegadores modernos.



La otra ventaja de JSON es que cualquier lenguaje de programación puede interpretarlo con facilidad. De hecho la mayoría de los lenguajes web trabajan nativamente con JSON.

ESTRUCTURA JSON

Como su nombre lo indica, **JSON** es muy similar a un **objeto literal**. La diferencias entre ellos son:

Objeto Literal

- Admite comillas simples y dobles
- Las claves del objeto van sin comillas
- Podemos escribir métodos
- Se recomienda poner una coma en la última propiedad

JSON

- Sólo se pueden usar comillas dobles
- Las claves van entre comillas
- No admite métodos, sólo propiedades y valores
- No podemos poner una coma en el último elemento

ESTRUCTURA JSON

JSON admite la mayoría de los tipos de datos de Javascript, veamos cómo sería la conversión entre ambos formatos.

VS

JS	JSON
<pre>{ texto: 'Mi texto', numero: 16, array: ['uno', 'dos'], booleano: true, metodo(): {return '¡Hola!'}, }</pre>	<pre>{ "texto": "Mi texto", "numero": 16, "array": ["uno", "dos"], "booleano": true }</pre>
⚠ JSON no soporta métodos	

Javascript nos proporciona un **objeto nativo** JSON con dos métodos que nos permiten convertir el **formato** de un archivo JSON a objeto literal o array, y viceversa.



JSON.parse()

Convierte un texto JSON al tipo de dato equivalente de Javascript

- **Recibe** una cadena de texto con formato JSON.
- **Devuelve** el mismo dato que recibió en formato Javascript.

```
let datosJson = '{"club": "Independiente", "barrio": "Avellaneda"}';
let datosConvertidos = JSON.parse(datosJson);
console.log(datosConvertidos);
// Se verá en consola un objeto literal
// {
//   club: 'Independiente',
//   barrio: 'Avellaneda'
// }
```

JSON.stringify()

Convierte un tipo de dato Javascript en un texto en formato JSON

- **Recibe** un tipo de dato de Javascript.
- **Devuelve** una cadena de texto con formato JSON

```
let objetoLiteral = { nombre: 'Carla', pais: 'Argentina' };
let datosConvertidos = JSON.stringify(datosObjeto);

console.log(datosConvertidos);
// Se verán en consola los datos en un string de tipo JSON
// '{ "nombre": "Carla", "pais": "Argentina" }'
```

7

8

Gracias a estos dos métodos
podremos generar un
formato transaccional de
fácil comprensión **entre**
distintos **sistemas**.



“

MÉTODOS DE STRINGS

Javascript nos ofrece varios **métodos y propiedades** para trabajar con los **strings**.



LOS STRINGS EN JAVASCRIPT

En muchos sentidos, para Javascript, un **string** no es más que un **array de caracteres**. Al igual que en los arrays, la primera posición siempre será 0.

```
var nombre = 'Fran';
    ↑↑↑
    0123
```

Así, podemos acceder al contenido de la variable nombre utilizando la misma sintaxis que con arrays: variable[indice]

```
nombre[3]; // devuelve 'n'
```

.length

Tal vez la conozcan de **arrays**. Esta propiedad retorna la **cantidad total** de los caracteres del string, incluidos los espacios.

Como es una propiedad, al invocarla no necesitamos los paréntesis.

```
var miSerie = 'Mad Men';
miSerie.length; // devuelve 7
```

```
var arrayNombres = ['Bart', 'Lisa', 'Moe'];
arrayNombres.length; // devuelve 3
```

¡SPOILER ALERT!

Antes de conocer los **métodos de los arrays**, es necesario espiar un poco la definición de **función** y de **método**, que veremos a fondo más adelante.

Una **función** es un bloque de código que nos permite ejecutar una tarea tantas veces como necesitemos. Las funciones pueden **recibir datos** para realizar la tarea y estos se conocen como **parámetros**.

Cuando una función le pertenece a un objeto, en este caso nuestro array, la llamamos **método**.

.indexOf()

Este método busca en el string donde se aplica, el string que recibe como parámetro.

En el caso de **encontrarlo**, retorna la primera **posición** donde lo encontró.

En el caso de **no encontrarlo**, retorna un **-1**.

```
var saludo = '¡Hola! Estamos programando';

saludo.indexOf('Estamos'); // devuelve 7
saludo.indexOf('vamos'); // devuelve -1, no lo encontró
saludo.indexOf('o'); // encuentra la letra 'o' que está
                     en la posición 2, devuelve 2 y corta la ejecución
```

.slice()

Este método **corta y devuelve una parte del string** donde se aplica.

Recibe 2 números como parámetros:

- El índice **desde donde** inicia el corte.
- El índice **hasta donde** hacer el corte y **es opcional**.

Ambos índices pueden recibir números negativos.

```
var frase = 'Breaking Bad Rules!';

frase.slice(9,12); // devuelve 'Bad'
frase.slice(13); // devuelve 'Rules!'
frase.slice(-10); // ¿Qué devuelve? ¡A investigar!
```

7

.trim()

Este método **elimina los espacios** que estén al principio y al final de un string.

Si hay espacios en el medio, no los quita.

No recibe parámetros.

```
var nombreCompleto = '    Homero Simpson    ';
nombreCompleto.trim(); // devuelve 'Homero Simpson'

var nombreCompleto = '    Homero    Simpson    ';
nombreCompleto.trim(); // devuelve 'Homero    Simpson'
```

8

.split()

Este método **divide un string** en varios strings utilizando el string que le pasemos como separador.

Devuelve un array con las partes del string original.

```
var cancion = 'And bingo was his name, oh!'
cancion.split(' ') //devuelve
['And', 'bingo', 'was', 'his', 'name', ',', 'oh!']
```

9

.replace()

Este método **reemplaza una parte de un string por otra**.

Recibe dos parámetros:

- El string que queremos buscar
- El string que usaremos de reemplazo

Devuelve un string nuevo con esa modificación.

```
var frase = 'Aguante Phyton!'
frase.replace('Phyton', 'JS') //devuelve 'Aguante JS!'
frase.replace('Phy', 'JS') //devuelve 'Aguante JSton!'
```

10

“

Si bien **cada método** realiza una **acción muy simple**.

Cuando los **combinamos**, podemos lograr **resultados** mucho **más complejos** y útiles.



OBJETOS LITERALES

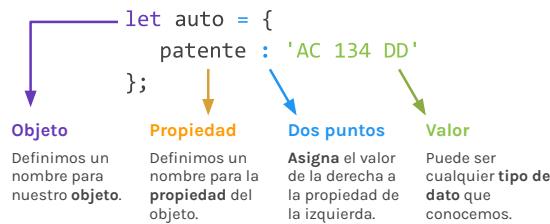
Podemos decir que son la **representación en código de un elemento de la vida real.**



ESTRUCTURA DE OBJETO LITERAL

Un **objeto** es una estructura de datos que puede contener **propiedades y métodos**.

Para crearlo usamos llave de apertura y de cierre {}.



3

ESTRUCTURA DE OBJETO LITERAL

Un **objeto** puede tener la cantidad de propiedades que queramos, si hay más de una las sepáramos con comas ,.

Con la notación **objeto.propiedad** accedemos al **valor** de cada una de ellas.

```
let tenista = {
  nombre: 'Roger',
  apellido: 'Federer'
};

console.log(tenista.nombre) // Roger
console.log(tenista.apellido) // Federer
```

MÉTODOS DE UN OBJETO

Una propiedad puede **almacenar** cualquier **tipo dato**.

Si una propiedad almacena una función, diremos que es un **método** del objeto.

```
let tenista = {
  nombre: 'Roger',
  apellido: 'Federer',
  edad: 38,
  saludar: function() {
    return '¡Hola! Me llamo Roger';
  }
};
```

MÉTODOS DE UN OBJETO

Para ejecutar el método de un objeto usamos la notación **objeto.método()**, los paréntesis del final son los que hacen que el método se ejecute.

```
let tenista = {
  nombre: 'Roger',
  apellido: 'Federer',
  saludar: function() {
    return '¡Hola! Me llamo Roger';
  }
};

console.log(tenista.saludar()) // ¡Hola! Me llamo Roger
```

5

6

TRABAJANDO DENTRO DEL OBJETO

La palabra reservada **this** hace referencia al objeto en sí desde el cual estamos invocando la palabra.

Con la notación **this.propiedad** accedemos al valor de **cada propiedad interna** de ese objeto.

```
let tenista = {  
    nombre: 'Roger',  
    saludar: function() {  
        return '¡Hola! Me llamo ' + this.nombre;  
    }  
};  
  
console.log(tenista.saludar()) // ¡Hola! Me llamo Roger
```

7

CONSTRUIR UN OBJETO

Javascript nos da una opción más para crear un objeto y es a través del uso de una **función constructora**.

La función constructora nos permite armar un molde y luego crear todos los objetos que necesitemos.

La función recibe un parámetro por cada propiedad que queramos asignarle al objeto.

```
function auto(marca, modelo){  
    this.marca = marca;  
    this.modelo = modelo;  
};
```

8

CONSTRUIR UN OBJETO

Objeto

Definimos un **nombre** para la función, que será el nombre de nuestro **objeto**. Por convención, solemos nombrar a los objetos con la primera letra mayúscula.



```
function auto(marca, modelo){  
    this.marca = marca;  
    this.modelo = modelo;  
};
```

9

CONSTRUIR UN OBJETO

Propiedades

Con la notación **this.propiedad** definimos la propiedad del objeto que estamos creando en ese momento.

Por lo general los valores de las propiedades serán los que vengan por parámetros.

```
function auto(marca, modelo){  
    this.marca = marca;  
    this.modelo = modelo;  
};
```

10

CONSTRUIR UN OBJETO

Parámetros

Definimos la cantidad de parámetros que consideremos necesarios para crear nuestro objeto.



```
function auto(marca, modelo){  
    this.marca = marca;  
    this.modelo = modelo;  
};
```

11

INSTANCIAR UN OBJETO

La función constructora Auto() espera dos parámetros: marca y modelo. Para crear un **objeto Auto** debemos usar la palabra reservada **new** y llamar a la función pasándole los parámetros que espera.

```
{ } let miAuto = new Auto('Ford', 'Falcon');
```

12

Cuando ejecutamos el método **new** para crear un objeto, lo que nos devuelve es una **instancia**. Es decir, en la variable **miAuto** tendremos almacenada una instancia del objeto Auto. Usando la misma función, podemos instanciar cuantos autos queramos.

```
{ } let miOtroAuto = new Auto('Chevrolet', 'Corvette');
```

8

10

12

ARROW FUNCTIONS

Las funciones son de lo que más vas a usar a la hora de programar en Javascript.

Las **arrow functions** nos permiten escribirlas con una **sintaxis** más **compacta**.



ESTRUCTURA BÁSICA

Pensemos en una función simple que podríamos programar de la manera habitual, una suma de dos números.

```
function sumar (a, b) { return a + b; }
```

Ahora veamos la versión reducida de esa misma función, al transformarla en una arrow function.

```
let sumar = (a, b) => a + b;
```

3

ESTRUCTURA BÁSICA

Parámetros

Usamos paréntesis para indicar los parámetros. Si nuestra función no recibe parámetros, debemos escribirlos igual.

```
let sumar = [(a, b)] => a + b;
```

Una particularidad de este tipo de funciones es que si recibe un único **parámetro**, podemos prescindir de los paréntesis.

```
let doble = [a] => a * 2;
```

5

ESTRUCTURA BÁSICA

Nombre

Las arrow functions, son **siempre anónimas**, es decir que no tienen nombre como las funciones normales.

```
(a, b) => a + b;
```

Si queremos nombrarlas, es necesario escribirlas como una **función expresada**, es decir, asignarla como valor de una variable.

```
let sumar = (a, b) => a + b;
```

4

ESTRUCTURA BÁSICA

Operador flecha

Lo usamos para indicarle a Javascript que vamos a escribir una función (reemplaza a la palabra reservada `function`).

Lo que está a la izquierda de la flecha será la entrada de la función (los parámetros) y lo que está a la derecha, la salida (el retorno)

```
let sumar = (a, b) => a + b;
```

6

ESTRUCTURA BÁSICA

Cuerpo

Escribimos la lógica de la función. Si la función tiene una sola línea de código y ésta misma es la que hay que retornar, no hacen falta las llaves ni la palabra reservada `return`.

```
let sumar = (a, b) => a + b;
```

De lo contrario, vamos a necesitar utilizar ambas.

```
let sumar = (a, b) => {  
    return a + b;  
};
```

“

Las **arrow functions** reciben su nombre por el operador `=>`

Si lo miramos con un poco de imaginación, se parece a una flecha.

En inglés suele llamarse **fat arrow** (flecha gorda) para diferenciarlo de otra combinación parecida ->



7

{ código }

```
let saludo = () => 'Hola Mundo!';
```

```
let dobleDe = numero => numero * 2;
```

```
let suma = (a, b) => a + b;
```

```
let horaActual = () => {  
    let fecha = new Date();  
    return fecha.getHours() + ':' + fecha.getMinutes();  
}
```

{ código }

```
let saludo = () => 'Hola Mundo!';
```

```
let dobleDe = numero => numero * 2;
```

```
let suma = (a, b) => a + b;
```

```
let horaActual = () => {  
    let fecha = new Date();  
    return fecha.getHours() + ':' + fecha.getMinutes();  
}
```

Arrow function sin parámetros.

Requiere de los paréntesis para iniciarse.

Al tener una sola línea de código, que esta misma sea la que quiero retornar, el `return` queda **implícito**.

9

10

{ código }

```
let saludo = () => 'Hola Mundo!';
```

```
let dobleDe = numero => numero * 2;
```

```
let suma = (a, b) => a + b;
```

```
let horaActual = () => {  
    let fecha = new Date();  
    return fecha.getHours() + ':' + fecha.getMinutes();  
}
```

Arrow function con un único parámetro (no necesitamos los paréntesis para indicarlo) y con un **return implícito**.

{ código }

```
let saludo = () => 'Hola Mundo!';
```

```
let dobleDe = numero => numero * 2;
```

```
let suma = (a, b) => a + b;
```

```
let horaActual = () => {  
    let fecha = new Date();  
    return fecha.getHours() + ':' + fecha.getMinutes();  
}
```

Arrow function con dos parámetros.

Necesita de los paréntesis y con un **return implícito**.

11

12

{ código }

```
let saludo = () => 'Hola Mundo!';

let dobleDe = numero => numero * 2;

let suma = (a, b) => a + b;

let horaActual = () => {
  let fecha = new Date();
  return fecha.getHours() + ':' + fecha.getMinutes();
}
```

Arrow function sin
parámetros y con un return
explicito.

En este caso hacemos uso de
las llaves y el return ya que la
lógica de esta función se
desarrolla en más de una
línea de código.

13

LOS CONDICIONALES IF TERNARIO SWITCH

1. IF TERNARIO

ESTRUCTURA DE UN IF TERNARIO

Condición

Declaramos una expresión que se evalúa como true o false.

```
4 > 10 ? 'El 4 es más grande' : 'El 10 es más grande'
```

“

Nos permiten **evaluar condiciones** y realizar diferentes acciones **según el resultado** de esas evaluaciones.



DEFINIENDO UN IF TERNARIO

A diferencia de un if tradicional, el **if ternario** se escribe de forma **horizontal**. Ambas estructuras tienen el mismo flujo interno (si esta condición es verdadera hacé esto, si no, hacé ésto otro) pero en este caso no hace falta escribir la palabra if ni la palabra else.



ESTRUCTURA DE UN IF TERNARIO

Primera expresión

Si la condición es **verdadera**, se ejecuta el código que está después del signo de interrogación.

```
4 > 10 ? 'El 4 es más grande' : 'El 10 es más grande'
```

ESTRUCTURA DE UN IF TERNARIO

Segunda expresión

Si la condición es **falsa**, se ejecuta el código que está después de los dos puntos.

```
4 > 10 ? 'El 4 es más grande' : 'El 10 es más grande'
```



Para el if ternario **es obligatorio poner código en la segunda expresión**, si no queremos que pase nada, podemos usar un string vacío".

7

2. SWITCH

DEFINIENDO UN SWITCH

El **switch**, al igual que los otros condicionales, evalúa una condición y, según el resultado, ejecuta sólo las líneas de código que correspondan.

Nos propone una **sintaxis más legible** para esos casos en los que queremos evaluar que un valor sea igual a alguno de los casos que proponemos.

También **nos permite agrupar casos** y ejecutar un mismo bloque de código para cualquier caso de ese grupo.

9

ESTRUCTURA DE UN SWITCH

El **switch** está compuesto por una **expresión** a evaluar, seguida de diferentes casos, tantos como queramos, cada uno contemplando un escenario diferente.

```
switch (expresión) {  
    case valorA:  
        //código que se ejecuta si valorA es verdadero  
        break;  
    case valorB:  
        //código que se ejecuta si valorB es verdadero  
        break;  
}
```

10

{ código }

```
let edad = 5  
  
switch (edad) {  
    case 10:  
        console.log('Tiene 10 años')  
        break;  
    case 5:  
        console.log('Tiene 5 años')  
        break;  
}
```

{ código }

```
let edad = 5  
  
switch (edad) {  
    case 10:  
        console.log('Tiene 10 años')  
        break;  
    case 5:  
        console.log('Tiene 5 años')  
        break;  
}
```

Definimos la variable **edad** y le asignamos el número 5.

11

12

{ código }

```
let edad = 5  
  
switch (edad) {  
    case 10:  
        console.log('Tiene 10 años')  
        break;  
    case 5:  
        console.log('Tiene 5 años')  
        break;  
}
```

Iniciamos el condicional con la palabra reservada **switch** y, entre paréntesis, la expresión / condición que queremos evaluar.

En este caso vamos a evaluar **qué valor tiene la variable edad**.

13

{ código }

```
let edad = 5  
  
switch (edad) {  
    case 10:  
        console.log('Tiene 10 años')  
        break;  
    case 5:  
        console.log('Tiene 5 años')  
        break;  
}
```

Por cada caso escribimos la palabra reservada **case** y a continuación el valor que queremos evaluar.

En este caso, **preguntamos si el valor de la variable edad es 10**.

Como este caso **NO es verdadero**, Javascript ignora el código de este caso y pasa a evaluar el siguiente caso.

14

{ código }

```
let edad = 5  
  
switch (edad) {  
    case 10:  
        console.log('Tiene 10 años')  
        break;  
    case 5:  
        console.log('Tiene 5 años')  
        break;  
}
```

Este caso **ES verdadero**, por lo tanto Javascript ejecuta el código que está asociado: en este caso un `console.log()`.

La palabra reservada **break** corta las siguientes evaluaciones. Si quisieramos que a pesar de ser verdadero este caso Javascript continúe evaluando los siguientes, simplemente no la escribimos.

15

¿Y SI NINGÚN CASO ES VERDADERO?

En ese caso introducimos la palabra reservada **default**, que nos permite tener un comportamiento por defecto cuando ningún caso evalúa como verdadero.

```
switch (expresión) {  
    case valorA:  
        // código que se ejecuta si valorA es verdadero  
        break;  
    default:  
        // código que se ejecuta si ningún caso es  
        // verdadero  
}
```

16

{ código }

```
let fruta = 'wefwef';  
  
switch (fruta) {  
    case 'manzana':  
        console.log('Qué rica es la manzana')  
        break;  
    case 'naranja':  
        console.log('¡Me encanta!')  
        break;  
    default:  
        console.log('No conozco esa fruta')  
        break;  
}
```

{ código }

```
let fruta = 'wefwef';  
  
switch (fruta) {  
    case 'manzana':  
        console.log('Qué rica es la manzana')  
        break;  
    case 'naranja':  
        console.log('¡Me encanta!')  
        break;  
    default:  
        console.log('No conozco esa fruta')  
        break;  
}
```

Definimos la expresión que vamos a evaluar en el **switch**.

En este caso queremos preguntar por el valor de la variable **fruta**.

17

18

{ código }

```
let fruta = 'wefwef';

switch (fruta) {
  case 'manzana':
    console.log('Qué rica es la manzana');
    break;
  case 'naranja':
    console.log('¡Me encanta!');
    break;
  default:
    console.log('No conozco esa fruta');
    break;
}
```

Este caso **es falso**, por lo tanto no se ejecuta su código.

{ código }

```
let fruta = 'wefwef';

switch (fruta) {
  case 'manzana':
    console.log('Qué rica es la manzana');
    break;
  case 'naranja':
    console.log('¡Me encanta!');
    break;
  default:
    console.log('No conozco esa fruta');
    break;
}
```

Este caso también **es falso**, por lo tanto no se ejecuta su código.

19

20

{ código }

```
let fruta = 'wefwef';

switch (fruta) {
  case 'manzana':
    console.log('Qué rica es la manzana');
    break;
  case 'naranja':
    console.log('¡Me encanta!');
    break;
  default:
    console.log('No conozco esa fruta');
    break;
}
```

Como ningún caso evaluó a verdadero, se ejecuta el código dentro del bloque **default**.

21

“

LOS CICLOS FOR

Los **ciclos** nos permiten **repetir instrucciones** de manera sencilla. Podemos hacerlo una determinada **cantidad de veces** o mientras que se **cumpla** una **condición**.



ESTRUCTURA DE UN CICLO FOR

Consta de **3 partes** que definimos dentro de los paréntesis. En conjunto, nos permiten determinar de qué manera se van a realizar las **repeticiones**, y definir las **instrucciones** que queremos que se lleven a cabo en cada una de ellas.

```
{ for (inicio; condición ; modificador) {
    //código que se ejecutará en cada repetición
}
```

{

3

ESTRUCTURA DE UN CICLO FOR

```
for (let vuelta = 1; vuelta <= 5; vuelta++) {
    console.log('Dando la vuelta número ' + vuelta);
};
```



En este ejemplo vamos a contar desde 1 hasta 5 inclusive:

```
● ● ●
Dando la vuelta 1
Dando la vuelta 2
Dando la vuelta 3
Dando la vuelta 4
Dando la vuelta 5
```

¡VEAMOS CADA PARTE EN ACCIÓN!

Inicio

Antes de arrancar el ciclo, se establece el **valor inicial** de nuestro contador.



```
for ( var vuelta = 1; vuelta <= 5; vuelta++ ) {
    console.log('Dando la vuelta número ' + vuelta);
};
```

¡VEAMOS CADA PARTE EN ACCIÓN!

Condición

Antes de ejecutar el código en cada vuelta, se pregunta si la condición resulta verdadera o falsa.

Si es **verdadera**, continúa con nuestras instrucciones.

Si es **falsa**, detiene el ciclo.

```
for ( var vuelta = 1; vuelta <= 5; vuelta++ ) {
    console.log('Dando la vuelta número ' + vuelta);
};
```

5

6

¡VEAMOS CADA PARTE EN ACCIÓN!

Modificador (incremento o decremento)

Luego de ejecutar nuestras instrucciones, se modifica nuestro **contador** de la manera que hayamos especificado, en este caso se le suma 1.

7

```
for ( var vuelta = 1; vuelta <= 5; vuelta++ ) {  
  console.log('Dando la vuelta número ' + vuelta);  
};
```

EL CICLO FOR EN ACCIÓN

En cada ciclo, se verifica si el valor de **vuelta** es menor o igual 5, si es así se ejecuta el **console.log()** y se incrementa el valor de **vuelta** en 1.

Cuando **vuelta** deje de ser menor o igual a 5, se corta el ciclo.

Iteración #	Valor de vuelta	¿ vuelta <= 5 ?	Ejecutamos
1	1	true	✓
2	2	true	✓
3	3	true	✓
4	4	true	✓
5	5	true	✓
6	6	false	✗

8

LOS CICLOS WHILE / DO WHILE

Los **ciclos** nos permiten **repetir instrucciones** de manera sencilla. Podemos hacerlo una determinada **cantidad de veces** o mientras que se **cumpla** una **condición**.



1. WHILE

ESTRUCTURA DEL CICLO WHILE

El ciclo while ejecutará nuestro código **mientras** la condición que especifiquemos se cumpla.

Consta de dos partes: la **condición** y el **bloque de código** que queremos ejecutar **mientras** que la condición establecida evalúe como **verdadera**.

```
while (condicion) {  
    // bloque de código que se ejecuta  
    // si la condición es verdadera  
}
```

4

{ código }

```
let contador = 0;  
  
while (contador <= 4) {  
    console.log('Contador: ' + contador);  
    contador++;  
}
```

{ código }

```
let contador = 0;  
  
while (contador <= 4) {  
    contador++;  
    console.log('Contador: ' + contador);  
}
```

Definimos la variable **contador** y le asignamos como valor el número 0.

{ código }

```
let contador = 0;  
  
while (contador <= 4) {  
    console.log('Contador: ' + contador);  
    contador++;  
}
```

Iniciamos el **ciclo** con la palabra reservada **while** y entre paréntesis, la expresión / condición que queremos evaluar.

Mientras sea verdadera, se ejecutará el código que definimos entre las llaves {}.

{ código }

```
let contador = 0;  
  
while (contador <= 4) {  
    console.log('Contador: ' + contador);  
    contador++;  
}
```

También vamos a mostrar por consola cuánto vale **contador** en ese momento.

7

8

{ código }

```
let contador = 0;  
  
while (contador <= 4) {  
    console.log('Contador: ' + contador);  
    contador++;  
}
```

En cada repetición, vamos a incrementar en 1 el valor que tenga asignada la variable **contador**.

{ código }

```
let contador = 0;  
  
while (contador <= 4) {  
    console.log('Contador: ' + contador);  
    contador++;  
}
```

En cada vuelta, se vuelve a evaluar la condición que definimos. Mientras sea verdadera, se ejecutará el bloque de código que definimos.

9

10

¿CÓMO SE VE EN CONSOLA?



2. DO WHILE

11

ESTRUCTURA DEL CICLO DO WHILE

A diferencia del while, el do while **primero ejecuta el código** y después evalúa la **condición**. Eso quiere decir que nuestro código **siempre** se ejecutará al menos **una vez**.

Al igual que el while, el do while ejecutará nuestro código mientras que la condición sea verdadera.

```
{  
    do {  
        // bloque de código que se ejecuta al menos una vez  
        // y cada vuelta si la condición es verdadera  
    } while (condicion)
```

{ código }

```
var diaSemana = 1;  
  
do {  
    console.log('Día de la semana N° ' + diaSemana);  
    diaSemana++;  
} while (diaSemana <= 7)
```

13

14

{ código }

```
var diaSemana = 1;  
  
do {  
    console.log('Día de la semana N° ' + diaSemana);  
    diaSemana++;  
} while (diaSemana <= 7)
```

Definimos la variable **diaSemana** y le asignamos el número 1.

{ código }

```
var diaSemana = 1;  
  
do {  
    console.log('Día de la semana N° ' + diaSemana);  
    diaSemana++;  
} while (diaSemana <= 7)
```

Iniciamos el **ciclo** con la palabra reservada **do**.

Escribimos entre llaves el código que queremos que se ejecute.

15

16

{ código }

```
var diaSemana = 1;  
  
do {  
    console.log('Día de la semana N° ' + diaSemana);  
    diaSemana++;  
} while (diaSemana <= 7)
```

El código que escribamos se ejecutará al menos una vez.

También se volverá a ejecutar mientras la condición sea verdadera.

{ código }

```
var diaSemana = 1;  
  
do {  
    console.log('Día de la semana N° ' + diaSemana);  
    diaSemana++;  
} while (diaSemana <= 7)
```

Finalizamos el **ciclo** con la palabra reservada **while**.

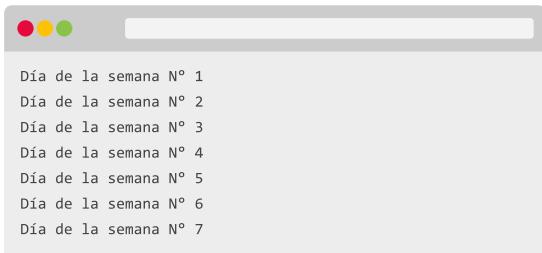
A continuación escribimos **entre paréntesis** la **condición** que queremos evaluar en cada vuelta.

El ciclo se repetirá **mientras ésta sea verdadera**

17

18

¿CÓMO SE VE EN CONSOLA?



19

CALLBACKS

Un **callback** es una **función** que se pasa como **parámetro** de otra **función**.

La función que lo recibe es quien se encarga de **ejecutarla** cuando sea necesario.



TIPOS DE CALLBACK

ANÓNIMO

En este caso la función que pasamos como **callback** no tiene nombre, es decir que es una **función anónima**.

Como las funciones anónimas no pueden ser llamadas luego por su nombre, necesitamos escribirla dentro de la función que se encargará de al callback.

```
setTimeout( function(){
  console.log('Hola Mundo!');
}, 1000)
```

TIPOS DE CALLBACK

DEFINIDO

La función que pasamos como **callback** puede ser una **función definida** previamente. Al momento de pasarla como parámetro de otra función, nos referiremos a la misma por su nombre.

```
let miCallback = () => console.log('Hola Mundo!');
setTimeout( miCallback, 1000);
```



Cuando enviamos una función como parámetro la escribimos **sin los parentesis**, ya que no queremos que se ejecute en ese momento. Será la función que la recibe quien se encargue de ejecutarla.

{ código }

```
function nombreCompleto(nombre, apellido) {
  return nombre + ' ' + apellido;
}

function saludar(nombre, apellido, callback) {
  return '¡Hola ' + callback(nombre, apellido) + '!';
}

saludar('Juanito', 'Sánchez', nombreCompleto);
```

{ código }

```
function nombreCompleto(nombre, apellido) {
  return nombre + ' ' + apellido;
}
```

```
function saludar(nombre, apellido, callback) {
  return '¡Hola ' + callback(nombre, apellido) + '!';
}
```

```
saludar('Juanito', 'Sánchez', nombreCompleto);
```

Definimos la función **nombreCompleto()**.

La misma se encarga de unir con un espacio un nombre y un apellido.

Nos **devuelve un string**.

{ código }

```
function nombreCompleto(nombre, apellido) {  
    return nombre + ' ' + apellido;  
}  
  
function saludar(nombre, apellido, callback) {  
    return '¡Hola ' + callback(nombre, apellido) + '!';  
}  
  
saludar('Juanito', 'Sánchez', nombreCompleto);
```

Definimos la función **saludar()**.

La misma recibe un nombre, un apellido y un **callback** como parámetros.

Este último será la función que vamos a querer ejecutar internamente.

{ código }

```
function nombreCompleto(nombre, apellido) {  
    return nombre + ' ' + apellido;  
}  
  
function saludar(nombre, apellido, callback) {  
    return '¡Hola ' + callback(nombre, apellido) + '!';  
}  
  
saludar('Juanito', 'Sánchez', nombreCompleto);
```

Lo que queremos devolver es un **string** completo.

La primera parte la tenemos en el **return**: '¡Hola (...)!.

El resto (...) vendrá de lo que nos devuelva el **callback** en el momento en el que se ejecute.

7

8

{ código }

```
function nombreCompleto(nombre, apellido) {  
    return nombre + ' ' + apellido;  
}  
  
function saludar(nombre, apellido, callback) {  
    return '¡Hola ' + callback(nombre, apellido) + '!';  
}  
  
saludar('Juanito', 'Sánchez', nombreCompleto);  
  
// Devolverá '¡Hola Juanito Sánchez!'
```

Ejecutamos la función **saludar**, le pasamos como parámetros un nombre, un apellido y la función **nombreCompleto**.

Primero se ejecutará el **callback**, que va a devolver el nombre completo y luego se ejecutará la función **saludar()**, que va a devolver el saludo completo.

{ código }

La función **saludar()** ¿sólo funciona si le pasamos como **callback** la función **nombre()**?

¡No!

Podemos pasarle cualquier función que devuelva un **string**, ya que en la estructura interna de **saludar()** definimos que opere con ese tipo de dato.



{ código }

```
function iniciales(nombre, apellido) {  
    return nombre[0] + apellido[0];  
}  
  
function saludar(nombre, apellido, callback) {  
    return '¡Hola ' + callback(nombre, apellido) + '!';  
}  
  
saludar('Juanito', 'Sánchez', iniciales);
```

Podríamos definir otra función que se encargue de devolvernos las iniciales del nombre y el apellido que nos pasen.

```
function iniciales(nombre, apellido) {  
    return nombre[0] + apellido[0];  
}
```

```
function saludar(nombre, apellido, callback) {  
    return '¡Hola ' + callback(nombre, apellido) + '!';  
}
```

```
saludar('Juanito', 'Sánchez', iniciales);
```

```
// Devolverá '¡Hola JS!'
```

Esta vez cuando ejecutamos la función **saludar()**, le pasamos la función **iniciales()** como **callback**.

Nuevamente se ejecutará el **callback**, esta vez va a devolver las iniciales y luego se ejecutará la función **saludar()**, que va a devolver el saludo completo.

11

12

“

MÉTODOS DE ARRAYS II

JavaScript nos provee de varios **métodos** para ejecutar sobre arrays, dándonos así un abanico de herramientas para poder trabajar con ellos.



¡SPOILER ALERT!

Antes de conocer esos métodos, es necesario espiar un poco la definición de **callback**.

Como ya sabemos, las funciones pueden recibir uno o más parámetros. En caso de recibirlas, éstos pueden ser, entre otros: un número, un string, un booleano o también... **¡una función!**

Cuando una método o función recibe a una **función como parámetro**, a esa función se la conoce como **callback**.

1. .map()

.map()

Este método recibe una función como parámetro (**callback**). Recorre el array y **devuelve un nuevo array** modificado. Las modificaciones serán aquellas que programemos en nuestra función de callback.

```
array.map(function(elemento){  
  // definimos las modificaciones que queremos  
  // aplicar sobre cada elemento del array  
})
```

{ código }

```
var numeros = [2, 4, 6];  
var dobleNumeros = numeros.map(function(num){  
  // Multiplicamos por 2 cada número  
  return num * 2;  
});  
  
console.log(dobleNumeros); // [4,8,12]
```

{ código }

```
var numeros = [2,4,6];
var dobleNumeros = numeros.map(function(num){
    return num * 2;
});

console.log(dobleNumeros); // [4, 8, 12]
```

Declaramos la variable `numeros` y almacenamos un array con tres números.

{ código }

```
var numeros = [2,4,6];
var dobleNumeros = numeros.map(function(num){
    return num * 2;
});

console.log(dobleNumeros); // [4, 8, 12]
```

Aplicamos el método `map` al array de números.

7

8

{ código }

```
var numeros = [2,4,6];
var dobleNumeros = numeros.map(function(num){
    return num * 2;
});

console.log(dobleNumeros); // [4, 8, 12]
```

Al `map()` le pasamos una función como parámetro (callback).

Esa función, a su vez, recibe un parámetro (puede tener el nombre que queramos).

El parámetro va a representar a cada elemento de nuestro array, en este caso, un número.

{ código }

```
var numeros = [2,4,6];
var dobleNumeros = numeros.map(function(num){
    return num * 2;
});

console.log(dobleNumeros); // [4, 8, 12]
```

Definimos el comportamiento interno que va a tener la función.

La función se va a ejecutar 3 veces: una por cada elemento de este array, y a cada uno lo va a multiplicar por 2.

9

10

{ código }

```
var numeros = [2,4,6];
var dobleNumeros = numeros.map(function(num){
    return num * 2;
});

console.log(dobleNumeros); // [4, 8, 12]
```

En la variable `dobleNumeros` vamos a almacenar el array que nos va a devolver el método `map`.

{ código }

```
var numeros = [2,4,6];
var dobleNumeros = numeros.map(function(num){
    return num * 2;
});

console.log(dobleNumeros); // [4, 8, 12]
```

Mostramos por consola la variable `dobleNumeros` que almacena un nuevo array con la misma cantidad de elementos que el original, pero con valores modificados.

11

12

2. .filter()

.filter()

Este método también recibe una función como parámetro. Recorre el array y **filtira** los elementos según una condición que exista en el callback.

Devuelve un **nuevo array** que contiene únicamente los elementos que hayan cumplido con esa condición. Es decir que nuestro nuevo array puede contener menos elementos que el original.

```
array.filter(function(elemento){  
    // definimos la condición que queremos utilizar  
    // como filtro para cada elemento del array  
});
```

14

{ código }

```
var edades = [22, 8, 17, 14, 30];  
var mayores = edades.filter(function(edad){  
    return edad > 18;  
});  
  
console.log(mayores); // [22, 30]
```

{ código }

```
var edades = [22, 8, 17, 14, 30];  
var mayores = edades.filter(function(edad){  
    return edad > 18;  
});  
  
console.log(mayores); // [22, 30]
```

Declaramos la variable edades y almacenamos un array con cinco números.

15

16

{ código }

```
var edades = [22, 8, 17, 14, 30];  
var mayores = edades.filter(function(edad){  
    return edad > 18;  
});  
  
console.log(mayores); // [22, 30]
```

Aplicamos el método filter al array de edades.

{ código }

```
var edades = [22, 8, 17, 14, 30];  
var mayores = edades.filter(function(edad){  
    return edad > 18;  
});  
  
console.log(mayores); // [22, 30]
```

Al método filter() le pasamos una función como parámetro (callback).

Esa función, a su vez, recibe un parámetro (puede tener el nombre que queramos).

El mismo va a representar a cada elemento de nuestro array, en este caso, una edad.

17

18

{ código }

```
var edades = [22, 8, 17, 14, 30];
var mayores = edades.filter(function(edad){
    return edad > 18;
});

console.log(mayores); // [22, 30]
```

Definimos el comportamiento interno que va a tener esa función.

La función se va a ejecutar 5 veces: una por cada elemento de este array, y los va a filtrar según la condición que definimos: que las edades sean mayores a 18.

Esto quiere decir que las que no cumplen con la condición (edad > 18 == false), serán excluidas.

{ código }

```
var edades = [22, 8, 17, 14, 30];
var mayores = edades.filter(function(edad){
    return edad > 18;
});
```

```
console.log(mayores); // [22, 30]
```

En la variable **mayores** almacené el array nuevo que me devuelve el método filter.

19

20

{ código }

```
var edades = [22, 8, 17, 14, 30];
var mayores = edades.filter(function(edad){
    return edad > 18;
});

console.log(mayores); // [22, 30]
```

Muestro por consola la variable **mayores** que tiene el array con los elementos que cumplieron con la condición establecida.

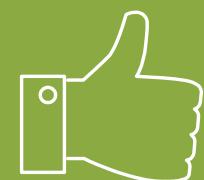
21

“

Es una **buena práctica** utilizar **nombres que tengan sentido** para nuestras **variables**.



```
var mayores = edades.filter(function(edad){
    return edad > 18;
});
```



De esa manera **queda más claro** para qué se van a utilizar.

3. **.reduce()**

.reduce()

Este método recorre el array y devuelve un **único valor**.

Recibe un callback que se va a ejecutar sobre cada elemento del array. El mismo, a su vez, recibe dos parámetros: un **acumulador** y el **elemento actual** que esté recorriendo.

```
{ }

array.reduce(function(acumulador, elemento){
    // definimos el comportamiento que queremos
    // implementar sobre el acumulador y el elemento
});
```

24

{ código }

```
var numeros = [5, 7, 16];
var suma = numeros.reduce(function(acum, num){
    return acum + num;
});

console.log(suma); // 28
```

{ código }

```
var numeros = [5, 7, 16];
var suma = numeros.reduce(function(acum, numero){
    return acum + numero;
});

console.log(suma); // 28
```

Declaramos la variable `numeros` y le asignamos un array con tres elementos.

25

26

{ código }

```
var numeros = [5, 7, 16];
var suma = numeros.reduce(function(acum, numero){
    return acum + numero;
});

console.log(suma); // 28
```

Le aplicamos el método `reduce()` al array de numeros.

{ código }

```
var numeros = [5, 7, 16];
var suma = numeros.reduce(function(acum, numero){
    return acum + numero;
});

console.log(suma); // 28
```

Al `reduce()` le pasamos una función como parámetro (callback).

Esa función recibe dos parámetros (pueden tener el nombre que queramos).

El primero va a representar el acumulador, el segundo el elemento que esté recorriendo en ese momento, en este caso un número.

27

28

{ código }

```
var numeros = [5, 7, 16];
var suma = numeros.reduce(function(acum, num){
    return acum + numero;
});

console.log(suma); // 28
```

Definimos el comportamiento interno de la función. En este caso, queremos devolver la **suma** total de los elementos.

El acumulador irá almacenando el resultado y por cada iteración sumará el elemento actual.

{ código }

```
var numeros = [5, 7, 16];
var suma = numeros.reduce(function(acum, num){
    return acum + numero;
});

console.log(suma); // 28
```

En la variable `suma` almacenamos lo que devuelva el método `reduce()` al aplicarlo al array `numeros`.

29

30

{ código }

```
var numeros = [5, 7, 16];
var suma = numeros.reduce(function(acum, num){
    return acum + numero;
});

console.log(suma); // 28
```

Mostramos por consola la variable **suma** que tiene la suma total de los números del array.

31

4.

.forEach()

.forEach()

La finalidad de este método es iterar sobre un array. Recibe un callback como parámetro y, a diferencia de los métodos anteriores, éste **no retorna nada**.

{ }

```
array.forEach(function(elemento){
    // definimos el comportamiento que queremos
    // implementar sobre cada elemento
});
```

{ código }

```
var paises = ['Argentina', 'Brasil', 'Colombia'];
paises.forEach(function(pais){
    console.log(pais);
});
```

33

34

{ código }

```
var paises = ['Argentina','Brasil','Colombia'];
paises.forEach(function(pais){
    console.log(pais);
});
```

Declaro la variable **paises** y le asigno un array con tres elementos.

35

{ código }

```
var paises = ['Argentina','Brasil','Colombia'];
paises.forEach(function(pais){
    console.log(pais);
});
```

Le aplico el método **forEach()** al array de **paises**.

36

{ código }

```
var paises = ['Argentina','Brasil','Colombia'];
paises.forEach(function(pais){
    console.log(pais);
});
```

Al método **forEach()** le pasamos una función como parámetro (callback).

Esa función recibe un parámetro, que va a representar a cada elemento del array, en este caso, a cada país.

{ código }

```
var paises = ['Argentina','Brasil','Colombia'];
paises.forEach(function(pais){
    console.log(pais);
});
```

Defino el comportamiento interno de la función.
En este caso, quiero mostrar por consola a cada país.

37

38

{ código }

```
var paises = ['Argentina', 'Brasil', 'Colombia'];
paises.forEach(function(pais){
    console.log(pais);
});
```

El método **forEach()** en este caso imprimirá por consola todos los elementos del array.



Argentina
Brasil
Colombia

39

“

Estas sentencias de Javascript nos van a permitir **iterar** elementos usando una **sintaxis clara y sencilla**.



FOR IN | FOR OF

1. FOR IN

ESTRUCTURA DEL FOR IN

El bucle `for ... in` nos permite **iterar** sobre cada una de las **propiedades** de un **objeto**.



{ código }

```
var persona = {  
    nombre: 'Guille',  
    edad: 23  
};
```

```
for (var dato in persona) {  
    console.log(dato, persona[dato]);  
};
```

Declaramos la variable **persona**, y almacenamos un objeto literal con las propiedades **nombre** y **edad**.

{ código }

```
var persona = {  
    nombre: 'Guille',  
    edad: 23  
};
```

```
for (var dato in persona) {  
    console.log(dato, persona[dato]);  
};
```

Declaramos la estructura del bucle `for ... in`.

{ código }

```
var persona = {  
    nombre: 'Guille',  
    edad: 23  
};  
  
for (var dato in persona) {  
    console.log(dato, persona[dato]);  
};
```

Declaramos la variable que va a representar a cada propiedad del objeto durante la iteración.
La misma puede declararse usando var o let y puede tener el nombre que queramos.

{ código }

```
var persona = {  
    nombre: 'Guille',  
    edad: 23  
};  
  
for (var dato in persona) {  
    console.log(dato, persona[dato]);  
};
```

Llamamos al objeto sobre el cual queremos iterar. En este caso, persona.

7

8

{ código }

```
var persona = {  
    nombre: 'Guille',  
    edad: 23  
};  
  
for (var dato in persona) {  
    console.log(dato, persona[dato]);  
};
```

Recordemos que por cada vuelta, a la variable dato se le asigna el **nombre de la propiedad** que está iterando en ese momento.
Usando el **nombre** del objeto y los **corchetes**, podemos acceder al **valor de cada propiedad**.

{ código }

```
var persona = {  
    nombre: 'Guille',  
    edad: 23  
};  
  
for (var dato in persona) {  
    console.log(dato, persona[dato]);  
};
```

Mostramos por consola el nombre y el valor de cada propiedad.

9

10

{ código }

```
for (var dato in persona) {  
    console.log(dato, persona[dato]);  
};
```

Así se verá por consola el nombre y el valor de cada propiedad.



2.
FOR OF

11

ESTRUCTURA DEL FOR OF

El bucle `for ... of` nos permite **iterar** sobre cada uno de los **valores** de un elemento iterable, por ejemplo, un array.

```
for(var variable of elementoIterable)
```

Palabra reservada
Iterador
Definimos una variable para referirnos a **cada ítem** del elemento que estamos iterando.

Palabra reservada
Elemento iterable
Elemento al cual queremos iterar para conocer su **valores**.

{ código }

```
var musicos = ['Charly', 'Spinetta', 'Fito'];
```

Declaramos la variable **musicos** y almacenamos un array con 3 elementos.

```
for (var musico of musicos) {  
    console.log(musico);  
};
```

13

14

{ código }

```
var musicos = ['Charly', 'Spinetta', 'Fito'];  
  
for (var musico of musicos) {  
    console.log(musico);  
};
```

Declaramos la estructura del **for ... of**.

{ código }

```
var musicos = ['Charly', 'Spinetta', 'Fito'];  
  
for (var musico of musicos) {  
    console.log(musico);  
};
```

Declaramos la variable que va a representar a cada elemento del array durante la iteración.

La misma puede declararse usando **var** o **let** y puede tener el nombre que queramos.

15

16

{ código }

```
var musicos = ['Charly', 'Spinetta', 'Fito'];  
  
for (var musico of musicos) {  
    console.log(musico);  
};
```

Llamamos al array **musicos** para iterar sobre él.

{ código }

```
var musicos = ['Charly', 'Spinetta', 'Fito'];  
  
for (var musico of musicos) {  
    console.log(musico);  
};
```

Mostramos por consola a cada músico.

17

18

{ código }

```
var musicos = ['Charly','Spinetta','Fito'];  
  
for (var musico of musicos) {  
  console.log(musico);  
};
```



Charly
Spinetta
Fito

Así se verá en consola el valor de cada elemento del array.

19

COMPARANDO SENTENCIAS

Para conseguir un objetivo puntual y elegir la mejor herramienta para hacerlo, es recomendable **siempre** pensar en el **contexto** y en las **características** de la herramienta que estamos por utilizar.

	FOR IN	FOR OF
Aplica para	Propiedades enumerables	Elementos iterables
¿La uso con objetos ?	✓	✗
¿La uso con arrays ?	⚠	✓
¿La uso con strings ?	⚠	✓

20



Javascript, al igual que muchos lenguajes de programación, nos ofrece un **objeto** para **generar fechas y trabajar con ellas**.

OBJETO DATE

CÓMO TRABAJAR CON DATE

Lo primero que tenemos que hacer para poder comenzar a trabajar con el objeto **Date** es crear una **instancia** del mismo.

```
var miFecha = new Date();
```

↓ ↓ ↓
Variable Palabra reservada Método constructor
Creo una variable para almacenar el objeto date.
Método que devuelve un objeto date de JavaScript.



Si no escribimos argumentos, el objeto Date se crea con la hora y fecha del momento.

1. MÉTODOS DE DATE

.getDate()

Este método retorna el **número del día del mes** de una fecha.
Devolverá un número entre **1** y **31**.

```
{ }  
var diaDeMiFecha = miFecha.getDate();  
console.log(diaDeMiFecha);  
// El N° del día de miFecha → ej: 22
```

.getMonth()

Este método retorna el **número del mes** de una fecha.
Devolverá un número entre **0** (enero) y **11** (diciembre).

```
{ }  
var mesDeMiFecha = miFecha.getMonth();  
console.log(mesDeMiFecha);  
// El N° del mes de miFecha → ej: 5 (Junio)
```

.getDay()

Este método retorna el **día de la semana** de una fecha.
Devolverá un número entre **0** (Domingo) y **6** (Sábado).

```
{ }  
  
var diaSemanaDeMiFecha = miFecha.getDay();  
console.log(diaSemanaDeMiFecha);  
// El N° de día de la semana de miFecha  
// → ej: 2 (Martes)
```

.getFullYear()

Este método retorna el **año completo** (4 dígitos) de una fecha.
Devolverá un número entre 1000 y 9999.

```
{ }  
  
var anioDeMiFecha = miFecha.getFullYear();  
console.log(anioDeMiFecha);  
// El N° del año actual → ej: 2019
```

7

8

CREANDO FECHAS DINÁMICAS

El objeto Date nos permite **crear** una fecha determinada. Al instanciar a nuestro objeto, podemos pasárle 3 parámetros que representan, en orden, el **año completo**, el **mes** y el **día**.

```
{ }  
  
var miFechaCumple = new Date(1983,11,24);
```

Ahora, la variable **miFechaCumple** es un **objeto** de tipo Date con una **fecha específica**, y le podemos implementar los **métodos** vistos anteriormente.

```
{ }  
  
miFechaCumple.getFullYear() //1983
```

9

DESTRUCTURING

Nos permite **extraer datos** de **arrays** y **objetos literales** de una manera más sencilla y fácil de implementar.



SIN USAR DESTRUCTURING

Para extraer datos de un **array**, es necesario crear una variable y asignarle un elemento del array usando el **operador de índice**.

```
0 let colores = ['Rojo', 'Azul', 'Amarillo'];
let azul = colores[1];
```

Para extraer datos de un **objeto**, es necesario que crear una variable y asignarle una **propiedad específica** de ese objeto.

```
0 let auto = {marca: 'Ford', anio: 1998};
let marcaAuto = auto.marca;
```

DESESTRUCTURANDO ARRAYS

Para desestructurar un **array**, declaramos una variable (**podemos usar var, let o const**), y entre **corchetes**, escribimos el nombre que queremos. Podemos declarar más de una variable, separando cada una con una coma **,**.

Luego igualamos esa estructura al array del cual queremos extraer los datos.

```
0 let colores = ['Rojo', 'Azul', 'Amarillo'];
let [rojo, azul, amarillo] = colores;
```

3

CÓMO FUNCIONA

Partiendo de un **array** previamente definido, se transfiere cada dato a las variables que definimos nosotros.

Javascript le asignará a cada variable el dato extraído de la estructura que elijamos, **respetando el orden original**.

```
0 let array = ['Rojo', 'Azul', 'Amarillo'];
let [color1, color2, color3] = array;
```

CÓMO FUNCIONA

Si queremos saltar un valor, podemos dejar vacío el nombre de la variable que correspondería con esa posición.

```
0 let array = ['Rojo', 'Azul', 'Amarillo'];
let [color1, , color2] = array;
```

5

6

DESESTRUCTURANDO OBJETOS

Para desestructurar un **objeto literal**, creamos una variable (*podemos usar var, let o const*), y entre **llaves**, declaramos el o los **nombres de las propiedades** que queremos extraer.

A esa estructura la igualamos al objeto del cual queremos extraer los datos.

```
let persona = {nombre: 'Laura', edad: 31, faltas: 3};  
let {nombre, edad} = persona;
```

CÓMO FUNCIONA

Partiendo de un objeto previamente definido, se transfiere cada propiedad o método a una o más variables que definamos.

Javascript le asignará a cada variable el valor de la propiedad que hayamos elegido.

```
let persona = {nombre: 'Laura', edad: 31, faltas: 3};  
let {nombre, faltas} = persona;
```

7

CÓMO FUNCIONA

Es posible que en algún caso necesitemos cambiarle el nombre a la variable que estamos creando.

En ese caso a continuación de la propiedad que estamos extrayendo colocamos dos puntos : seguidos del nuevo nombre.

```
let persona = {nombre: 'Laura', edad: 31, faltas: 3};  
let {nombre, faltas: totalFaltas} = persona;
```

9

La desestructuración **no modifica** el array u objeto literal de origen.



Su único objetivo es **copiar** los **valores** de una manera más práctica y rápida.

SPREAD OPERATOR REST PARAMETER

“

Este operador permite **expandir** cada uno de los datos de un **elemento iterable** dentro de otro elemento.



1.

SPREAD OPERATOR

USO Y SINTAXIS

El operador de **propagación** se puede usar sobre cualquier elemento iterable. Nos sirve para copiar y mover datos de un lugar a otro de una forma eficaz.



SPREAD EN ARRAYS

Implementando este operador, podemos **copiar** todos los datos de un array en un **array nuevo**.

```
let clubesUno = ['Boca', 'River', 'Racing'];
let clubesDos = ['San Lorenzo', 'Lanús', 'Gimnasia'];
let todosLosClubes = [...clubesUno, ...clubesDos];
```

También podemos **agregar** todos los datos de un array **dentro** de un array existente.

```
let parte = ['los', 'cumplas'];
let oracion = ['Que', ...parte, 'feliz'];
```

SPREAD EN OBJETOS

Implementando este operador, podemos **copiar** todas las propiedades de un objeto **dentro** de otro **objeto existente**.

```
let auto = {marca:'Ferrari', kms:0, anio:2019};
let corredorUno = {nombre:'Vettel', edad:32, ...auto};
let corredorDos = {nombre:'Leclerc', edad:21, ...auto};
```

Tanto `corredorUno` como `corredorDos` ahora tienen todas las propiedades que definimos en el objeto `auto` sin tener que definirlas a mano en cada uno de ellos.

SPREAD Y FUNCIONES

Implementando este operador, podemos pasarle a una función un **array** como **argumento**. El operador `...` se encargará de expandir los datos para que la función los tome como argumentos separados.

Para exemplificar usaremos el método de JS `Math.min()` que recibe N cantidad de argumentos y devuelve el menor.

```
let notas = [9.3, 8.5, 3.2, 7, 10];
Math.min(...notas); // Devuelve 3.2
```

2.

REST PARAMETER

7

“

Utilizado como **último parámetro** de una función nos permite **capturar** cada uno de los **argumentos adicionales** pasados a esa **función**.



EL PARÁMETRO REST

El **parámetro rest** se escribe de la misma manera que el **operador spread** `...`. La diferencia es que se utiliza durante la definición de la función y no durante su ejecución.

El parámetro rest **generará un array** con todos los **argumentos adicionales** que se le pasen a la función.

```
function miFuncion(param1, param2, ...otros) {
  return otros;
}
miFuncion('a', 'b', 'c', 'd', 'e');
// retornará ['c', 'd', 'e']
```

{}

EL PARÁMETRO REST

Implementando el parámetro rest, podemos **definir** una función que acepte cualquier número de argumentos.

```
function sumar(...numeros) {
  // Sabiendo que numeros es ahora un array utilizamos
  // el método reduce para obtener la sumatoria
  return numeros.reduce((acum, num) => acum += num);
}

sumar(1, 4); // devuelve 5
sumar(13, 6, 8, 12, 23, 37); // devuelve 99
```

{}

EL PARÁMETRO REST



Como el **parámetro rest** captura todos los argumentos restantes, **siempre debe ser el último parámetro de la función**, de lo contrario, recibiremos un error.

```
function sumar(...numeros, otroParámetro) {
  // Utilizamos el método reduce para obtener la suma
  return numeros.reduce((acum, num) => acum += num);
}

SyntaxError: parameter after rest parameter
```

11

10

TRABAJAR CON UN REPOSITORIO

Los siguientes **comandos** son el **paso a paso** que necesitamos para **trabajar** con un **repositorio**.



COMANDOS PASO A PASO

```
>_ git init // crea el repositorio  
>_ git config user.name "nombreUsuario" // agrega nuestra identidad  
>_ git config user.email "emailUsuario" // agrega nuestra identidad  
>_ git remote add origin http://... // apunta al repositorio remoto  
>_ git add . // agrega todos los cambios
```

COMANDOS PASO A PASO

```
>_ git commit -m "mensaje" // commitea los cambios hechos  
>_ git push origin master // envía los cambios al repositorio remoto
```

3

4

Siempre viene bien tener a mano la **documentación**, para ello podemos visitar:

<http://dev.to/t/git>
<http://ohshitgit.com>
<http://git-scm.com>





>_índice

- >_ ¿Qué es GIT?
- >_ ¿Por qué usar un sistema de control de versiones?
- >_ Instalación de GIT
- >_ ¿Qué es GitHub?
- >_ ¿Qué es un repositorio?
- >_ Tipos de repositorios
- >_ Crear un Repositorio Remoto (GitHub)
- >_ Crear un Repositorio Local (PC)
- >_ Agregar nuestra identidad al Repositorio Local
- >_ Conectar Repositorio Local con Repositorio Remoto
- >_ ¿Qué es un commit?
- >_ ¿Qué significa que un archivo está en seguimiento?
- >_ Subiendo archivos a un Repositorio Remoto
- >_ Actualizando archivos de un Repositorio Local
- >_ Clonando archivos de un Repositorio Remoto

>_

¿Qué es GIT?

Es un **software de control de versiones** que **registra** los **cambios** realizados sobre un archivo o conjunto de archivos a lo largo del **tiempo**. De esta forma, podemos recuperar y tener **acceso** a versiones específicas cuando queramos.

>_

¿Por qué usar un sistema de control de versiones?

Usar un sistema de control de versiones (VSC), te permite **revertir archivos** y **proyectos enteros** a un estado anterior, **comparar cambios** a lo largo del tiempo, ver **quién modificó** por última vez algo que puede estar causando un problema, quién introdujo un error y cuándo, y mucho más.

>_

Instalación de GIT

- ⇒ Ir a la [web oficial](#) y descargar el ejecutable
- ⇒ Ejecutar el archivo que descargamos
- ⇒ Si tu sistema operativo es Windows, además de instalarse GIT, se instalará en tu máquina una terminal llamada **GIT BASH**
- ⇒ Una vez instalado GIT, estará disponible el comando `git` para correr en la terminal
- ⇒ Para verificar que la instalación se haya realizado correctamente, abrir una terminal y correr el comando `git --version`

>_

¿Qué es GitHub?

GitHub es un **sitio web** en donde podemos **almacenar los archivos y proyectos** de programación de manera **gratuita**. Para poder hacer uso de sus beneficios sólo hace falta [crearse una cuenta](#) en la plataforma.

>_

¿Qué es un repositorio?

Es el lugar donde se irán **almacenando** los **archivos** de nuestro proyecto. En GitHub podemos tener la cantidad de proyectos que queramos, en donde a **cada proyecto** le corresponderá **un repositorio**.



Tipos de repositorios

Los repositorios que se alojan en GitHub los llamamos **Repositorios Remotos**, mientras que a los que se alojan en nuestra PC los llamamos **Repositorios Locales**. Es necesario crear un **vínculo** entre ambos para poder mantener **actualizados** los archivos locales que están conectados a ese repositorio en la nube.



Crear un Repositorio Remoto (GitHub)

- ⇒ Una vez iniciada la sesión en GitHub, dar click al ícono **+** sobre la barra principal y elegir la opción **New repository**
- ⇒ Veremos en pantalla un formulario
- ⇒ Completar únicamente el nombre que le queremos dar a nuestro repositorio. Podemos nombrarlo como queramos, pero debe ser un nombre que no hayamos usado para otro repositorio
- ⇒ Ir hacia abajo de todo y apretar el botón **Create Repository**



Crear un Repositorio Local (PC)

- ⇒ Crear una carpeta en nuestra PC, que será donde alojaremos nuestro proyecto. Esta carpeta será nuestro **repositorio local**
- ⇒ Dentro de la carpeta, abrir una terminal y correr el comando **git init**
- ⇒ Este comando **inicializa** un **repositorio local** en la carpeta del proyecto



Agregar nuestra identidad al Repositorio Local

Para que git pueda hacer un **completo seguimiento** de los **cambios** realizados, necesitamos decirle al repositorio quien somos.

- ⇒ Abrir una terminal en la ubicación de nuestro repositorio local
- ⇒ Correr el comando `git config user.name "nombreDeUsuario"` en donde, entre comillas, debemos escribir nuestro nombre de usuario tal cual aparezca en GitHub
- ⇒ Para verificar que ingresamos bien nuestro nombre de usuario, correr el comando `git config user.name` y presionar enter
- ⇒ Correr el comando `git config user.email "nombre@email.com"` en donde, entre comillas, debemos escribir el email con el que nos registramos en GitHub
- ⇒ Para verificar que ingresamos bien nuestro email, correr el comando `git config user.email` y presionar enter

Para configurar nuestra identidad de manera global y no tener que estar aclarando siempre nuestro email y nombre de usuario, agregar la palabra `--global`

- ⇒ `git config --global user.name "nombreDeUsuario"`
- ⇒ `git config --global user.email "nombre@email.com"`



Conectar Repositorio Local con Repositorio Remoto

Para que nuestro Repositorio Local sepa a donde queremos subir nuestros archivos hace falta especificarlo.

- ⇒ Tener creado previamente un repositorio en GitHub
- ⇒ Ir a la ubicación del mismo y copiar la url
- ⇒ Escribir el comando `git remote add origin`
- ⇒ Pegar la **url** después de la palabra `origin` (dejando un espacio de por medio) y presionar enter
- ⇒ Para verificar que el paso anterior se ejecutó correctamente, correr el comando `git remote -v`. Deberías ver en la terminal la palabra `origin` seguida de la url



¿Qué es un commit?

Cada vez que subimos archivos (nuevos o modificados) a un Repositorio Remoto, se suben en forma de un pequeño **paquete de archivos**. Cada paquete tiene una **fecha de creación** (timestamp) y un **autor**.

Es a través de los **commits** que vamos a poder hacer el seguimiento de los cambios que se van realizando en los proyectos, ya que cada uno de ellos genera un **punto cronológico** en la línea del tiempo del proyecto.



¿Qué significa que un archivo está en seguimiento?

Cuando enviamos un archivo al repositorio, estamos diciéndole a GIT que queremos hacerle un **seguimiento** al mismo a través del tiempo.

Es decir, queremos que se guarde el **estado actual** de ese archivo, para que cada vez que hagamos un cambio nuevo y lo envíemos, podamos **comparar estados** y ver cómo estaba en determinado momento. **Seguirlo** a lo largo del proyecto.



Subiendo archivos a un Repositorio Remoto

Para poder subir nuestros archivos a la nube, debemos seguir los siguientes pasos:

- ⇒ Abrir una terminal en la ubicación de nuestro repositorio local
- ⇒ Correr el comando `git status` para ver el **estado** de nuestros archivos (aquellos en rojo son los archivos que aún no están en seguimiento)
- ⇒ Correr el comando `git add .` para indicar que queremos **agregar todos** los archivos al repositorio
- ⇒ Para **agregar un sólo archivo**, correr el comando `git add archivo.extension` en donde deberemos indicar tanto el nombre como la extensión del archivo
- ⇒ Correr el comando `git status` para ver el **estado** de nuestros archivos (aquellos en verde son los archivos que serán agregados al repositorio, por lo tanto estarán en seguimiento)
- ⇒ Para **confirmar** que queremos subir de manera definitiva aquellos archivos que agregamos, correr el comando `git commit -m "mensaje"` en donde, entre comillas, deberemos escribir, en lo posible, un mensaje corto que resuma el trabajo que estamos subiendo
- ⇒ Para **enviar** los archivos al repositorio remoto correr el comando `git push origin master`

>_

Actualizando archivos de un Repositorio Local

Para poder actualizar los archivos de nuestro repositorio local con respecto a los que están en el repositorio remoto, debemos correr el comando `git pull origin master`

>_

Clonando archivos de un Repositorio Remoto

Para **descargar** por primera vez un repositorio remoto a nuestra máquina, tendremos que **clonarlo**.

- ⇒ Abrir una terminal en la ubicación en donde queramos clonar el proyecto
- ⇒ Copiar la **url** del repositorio que queremos clonar
- ⇒ Escribir el comando `git clone`
- ⇒ Pegar la **url** después de la palabra `clone` (*dejando un espacio de por medio*) y presionar enter



INTRODUCCIÓN A METODOLOGÍAS ÁGILES Y SCRUM

ANTES DE
HABLAR DE
AGILIDAD...
VAMOS A
HACER UNA
¡ACTIVIDAD!



**CONSTRUYAMOS
UN BARCO...**

AHORA, UN POCO
DE **HISTORIA...**

¿QUÉ PASABA
ANTES DE LAS
METODOLOGÍAS
ÁGILES?



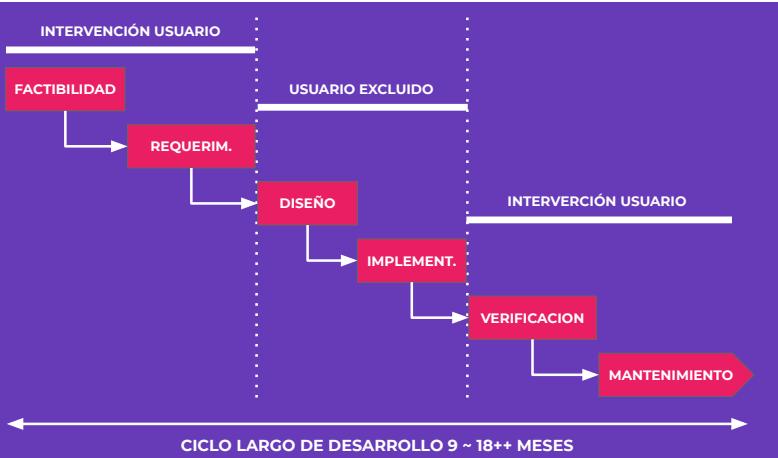
**METODOLOGÍAS
DE CASCADA**

En 1970 Winston Royce publica un artículo sobre cómo estructurar proyectos grandes y complejos, basado en sus experiencias en la NASA.

En 1985 el DoD (Departamento de Defensa de EEUU) requiere que todo desarrollo de software siga el procedimiento delineado por Winston.

A partir de ese momento, la metodología de cascada se convierte en estándar para el desarrollo de software.





LO BUENO

Procesos mecánicos

Funcionan muy bien para procesos donde no hay incertidumbre.

Fases y costos

Al encontrarse con procesos mecánicos en terrenos con baja incertidumbre, las fases y los costos están bien calculados.

LO MALO



Cambios

Tienen poca capacidad de adaptación a los cambios de requerimiento.



Requerimientos

Deben conocerse todos los requerimientos desde el momento 0 del proyecto.



Retrasos

Al estar tan marcada cada etapa, si una se retrasa, traslada el retraso a la próxima y todas las demás consecutivamente.

¿CÓMO LLEGAMOS DEL PUNTO A AL PUNTO B?

A ————— B



MANIFIESTO ÁGIL (2001)

¡ES HORA DE CAMBIAR!

Un compromiso público para conseguir mejores formas de trabajo y ayudar a otros a implementarlas.

Contrapuesta a los estándares del Product Manager Institute.

Consta de 12 principios y 4 valores principales.

<http://www.agilemanifesto.org>



MANIFIESTO ÁGIL (2001)

Todos estos valores son importantes, pero vamos a darle prioridad a los de la izquierda.

Personas e interacciones
Software funcional
Colaboración con el cliente
Responder ante el cambio

Procesos y herramientas
Documentación compleja
Apegio a contratos
Seguimiento de un plan



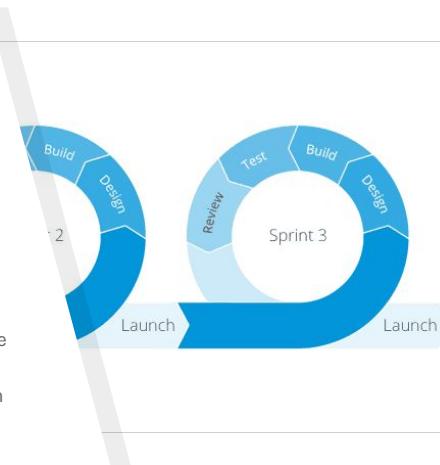
METODOLOGÍAS ÁGILES



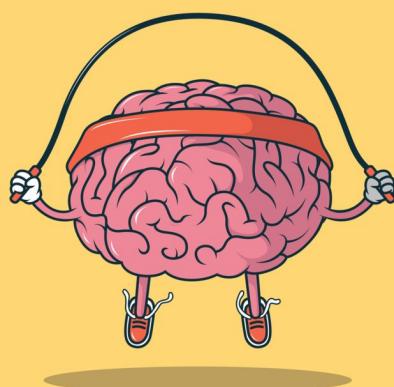
METODOLOGÍAS ÁGILES

SON TÉCNICAS QUE NOS PERMITEN:

- Flexibilidad
- Reducción de frustraciones
- Reducción de tiempos
- Reducción de costos
- Entregar valor constantemente
- Hacer énfasis en las personas
- Fomentar la auto-organización



LAS METODOLOGÍAS ÁGILES REQUIEREN UN CAMBIO DE PENSAMIENTO

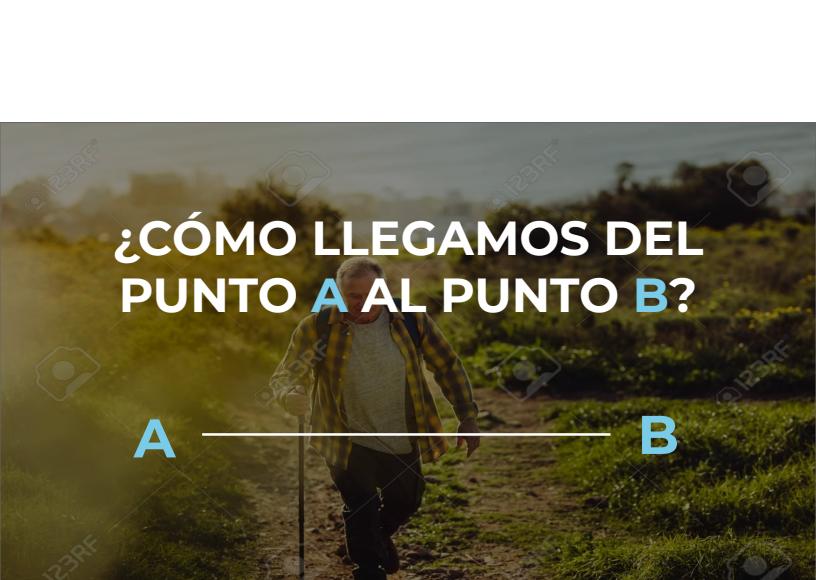


LAS METODOLOGÍAS ÁGILES SON ITERATIVAS



¿CÓMO LLEGAMOS DEL PUNTO A AL PUNTO B?

A ————— B



SCRUM



TOMÉMOSNOS UN DESCANSITO



Product Owner

PRODUCT OWNER

Representa la **voz del cliente**. Es quien toma las decisiones finales del producto.

Se asegura de que el equipo **trabaje de forma adecuada** desde la perspectiva del negocio.

Escribe historias de usuario, las **prioriza**, y las coloca en el Product Backlog.

23



Scrum Master

SCRUM MASTER

Es un **facilitador**.

Responsable de controlar el tiempo, las conversaciones y el proceso en general.

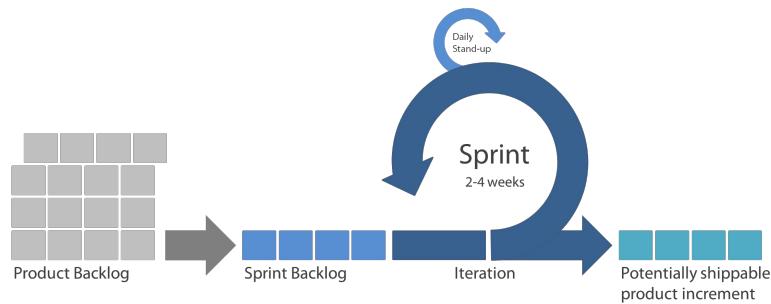
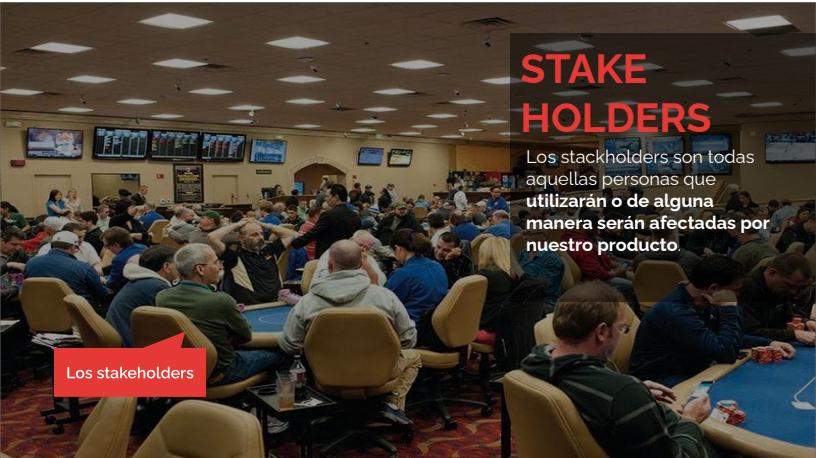
Ayuda a **eliminar obstáculos**.

Actúa como **protección entre el equipo y distracciones**.

Comprueba que cada miembro del equipo **utilice sus habilidades únicas para el éxito** del equipo.

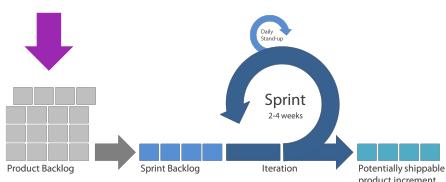
No es el jefe del equipo.

24



BACKLOG DEL PRODUCTO

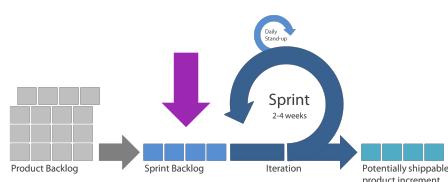
- Es el conjunto de todos los requisitos de proyecto
- Contiene descripciones genéricas de funcionalidades deseables, priorizadas según su retorno de inversión (ROI).
- Representa la totalidad de lo que va a ser construido.



29

BACKLOG DEL SPRINT

- Es el subconjunto de requisitos que serán desarrollados durante el sprint actual.

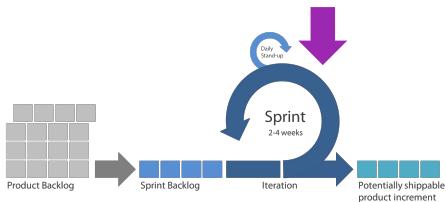


30

SPRINT

› El Sprint es el período en el cual se lleva a cabo el trabajo en sí.

› Su duración es constante y definida por el equipo.



31

HISTORIA DE USUARIO / USER STORY

› Se definen en base a la empatía.

› Deben ser específicos de acuerdo a la prioridad.

› Deben tener criterios de aceptación.



32

TABLERO DE TAREAS

› **Story:** Historias de usuario que originaron las tareas.

› **To Do:** las tareas a realizar en el sprint.

› **En progreso:** las tareas en curso en este sprint.

› **A verificar:** tareas listas que requieren verificación

› **Terminadas:** tareas listas y verificadas (criterios de aceptación).



33

CEREMONIAS DE SCRUM



PLANIFICACIÓN

› Definir qué trabajo se hará.

› Esta ceremonia se realiza con el **equipo completo**, donde se acuerda sobre el trabajo para hacer durante el sprint.

› Identificar y comunicar **cuánto esfuerzo** es probable que se tenga que invertir en las distintas tareas que se proponen.

› Dedicar a la planificación **8 hrs como tiempo límite** (sprint de 1 mes).

35

STAND-UP / SCRUM DAILY

› Reuniones diarias que se realizan **todos de pie**

› **15 minutos**

› **3 preguntas claves**

- ¿Qué hiciste ayer?
- ¿Tuviste impedimentos para lograr tus objetivos?
- ¿Qué vas a hacer hoy?

36

DEMO / REVISIÓN

- Revisar el trabajo que fue completado y el que no.
- Presentar el trabajo completado a los stakeholders.
- El trabajo incompleto no puede ser demostrado.
- La reunión no debe durar más de 4 hrs (sprint de 1 mes).

37

RETROSPECTIVA

- El equipo deja sus **impresiones respecto al sprint** recién superado.
- El propósito es realizar la **mejora continua del proceso y del equipo**.
- La **duración** de esta reunión es de **4 horas fijas** (sprint 1 mes).

38

ARQUITECTURA CLIENTE SERVIDOR

Dentro del contexto de desarrollo web, esta arquitectura hace referencia a un **modelo de comunicación** que vincula a varios dispositivos con un servidor a través de **internet**.



A QUÉ LLAMAMOS CLIENTE

Son los **dispositivos que hacen peticiones** de servicios o recursos a un **servidor**.

Pueden ser: una computadora, un teléfono celular, una tablet, una consola de video juegos o cualquier implemento que tenga la capacidad de conectarse a una **red**.

Dentro de Internet, el cliente suele acceder a estos servicios y recursos a través de un **navegador web**.



A QUÉ LLAMAMOS SERVIDOR

Es el **equipo** que **brinda los servicios y recursos** a los que acceden los clientes.

Es importante tener en cuenta que la misma computadora puede ser el **cliente** y el **servidor** al mismo tiempo.

De hecho es lo más normal en el entorno de desarrollo de un sitio o aplicación web.



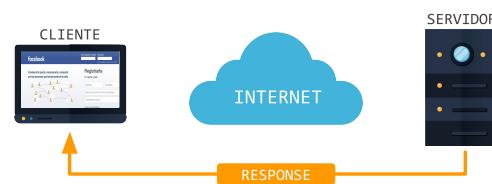
FLUJO CLIENTE SERVIDOR



REQUEST / SOLICITUD

Es la **solicitud** que hacemos a través del navegador (el cliente) a un servidor, en este ejemplo, la página de Facebook que está almacenada en sus servidores.

FLUJO CLIENTE SERVIDOR



RESPONSE/ RESPUESTA

El servidor recibe nuestra solicitud, la **procesa**, y envía como resultado una **respuesta** al cliente (navegador), en este ejemplo devolverá la página principal del sitio.

¿Por qué es **importante** conocer este flujo Request-Response?

Porque dentro del mundo del desarrollo web, la mayoría de las aplicaciones tienen dos claros frentes: el **frontend** y el **backend**.



FRONT-END

Es todo lo que pasa del lado del **cliente** (en el navegador).

Aquí se incluyen todos los elementos gráficos que conforman la interfaz del sitio.

Los lenguajes que se manejan son **HTML**, para la estructura, **CSS**, para los estilos visuales y **Javascript**, para la interacción dentro del sitio.



BACK-END

Es todo lo que pasa del lado del **servidor**.

Aquí se incluye todo el funcionamiento interno y lógica del sitio. Es lo que permite que se carguen todas las peticiones solicitadas por el cliente.

Algunos de los lenguajes que se manejan son **MySQL**, para base de datos, **PHP**, para sitios webs dinámicos, entre otros.



Hoy existe la posibilidad de correr Javascript del lado del **servidor**, permitiéndonos programar en un mismo lenguaje tanto en **front** como en **back**, logrando que el proceso de **desarrollo** sea más **fluido**.



“

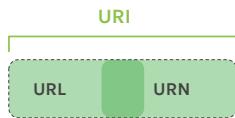
INTRODUCCIÓN A HTTP

HTTP (*Hyper Text Transfer Protocol*) es el **protocolo** que gestiona las **transacciones** web entre clientes y servidores.



QUÉ ES UNA URI

El protocolo HTTP permite la transferencia de información en la web a través de **direcciones web**, técnicamente llamadas **URI**. Una **URI** (*identificador de recursos uniformes*) es un bloque de texto que se escribe en la barra de direcciones de un navegador web y está compuesto por dos partes: la **URL** y la **URN**.



3

COMPONENTES DE UNA URI

URL

Indica **dónde** se encuentra el recurso que deseamos obtener y siempre comienza con un **protocolo**. En este caso HTTP.



4

COMPONENTES DE UNA URI

URN

Es el **nombre exacto** del **recurso** uniforme. El nombre del dominio y, en ocasiones, el nombre del recurso.



5

Dentro de esta estructura de comunicación, hablamos de **request** cada vez que el cliente le solicita un recurso al servidor, y de **response** cada vez que el servidor le devuelve una respuesta al cliente.



CÓMO VIAJA LA INFORMACIÓN

Cada vez que usamos este protocolo, se envía a través de él información importante. La información viaja a través de los **headers o cabeceras**, que son **porciones de texto** conteniendo esa información requerida por el cliente y por el servidor.



7

MÉTODOS DE PETICIÓN

El protocolo HTTP define **métodos de petición**. Cada método representa una **acción** y, si bien comparten algunas características, implementan funcionalidades diferentes entre sí. Los métodos más utilizados por este protocolo son:

GET

Se utiliza para **pedirle información** al servidor de un recurso específico. Cada vez que escribimos una dirección en el navegador o accedemos a un enlace, estamos utilizando el método GET. En caso de querer **enviar información** al servidor usando éste método, la misma viajará a través de la **URL**.

MÉTODOS DE PETICIÓN

POST

Se utiliza para **enviar datos** al servidor. Este método es más seguro que get ya que la información no viaja a través de la URL.

DELETE

Borra un recurso presente en el servidor. Cuando eliminamos un posteo en facebook, por ejemplo, estamos utilizando este método.

9

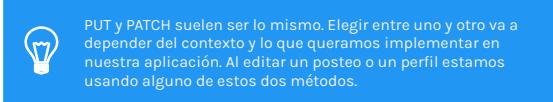
MÉTODOS DE PETICIÓN

PUT

Es muy parecido a post. Se usa para **reemplazar** toda la **información** actual de un recurso presente en el servidor.

PATCH

Similar a PUT. Es utilizado para aplicar **modificaciones parciales** a un recurso en el servidor.



CÓDIGOS DE ESTADO HTTP

Cada vez que el **servidor** recibe una petición **request**, el mismo emite un código de estado que indica, de forma abreviada, el **estado** de la respuesta HTTP. El código tiene tres dígitos. El primero representa uno de los 5 tipos de respuesta posibles:

- 1xx_ Respuestas informativas
- 2xx_ Respuestas exitosas
- 3xx_ Redirecciones
- 4xx_ Errores del cliente
- 5xx_ Errores de servidor

Algunos **códigos** conocidos:

200 OK

307 Temporary Redirect

403 Forbidden

404 Not Found

500 Internal Server Error



HTTPS es un protocolo mejorado de **HTTP**. Usando este protocolo, el servidor **codifica** la sesión con un **certificado digital**, dándole al usuario ciertas garantías de que la información que envíe desde esa página no será **interceptada** ni **utilizada** por terceros.



HTTP EN NODE JS

Usando el módulo nativo HTTP, podemos **crear** un **servidor web** dentro de nuestro proyecto.



CÓMO USAR HTTP

Lo primero que tenemos que hacer es requerir el módulo nativo en el **entry-point** de nuestra aplicación: `app.js`.

```
{ const http = require('http');
```

En la variable `http` tenemos almacenado un objeto, que presenta todas las propiedades y funcionalidades que necesitamos para crear nuestro servidor. Lo siguiente, es pedirle a ese objeto, el método `createServer()` encargará de levantar el servidor y manejar las peticiones que le lleguen.

```
{ http.createServer();
```

CÓMO USAR HTTP

Es momento de definir el puerto en el que el servidor escuchará las peticiones. Eso lo haremos a través del método `listen()`, el cual recibe dos parámetros: el primero, el **puerto** donde se escuchará la aplicación (puede ser cualquier número de 4 dígitos), y el segundo, el **dominio** donde queremos que se ejecute el servidor.

```
{ http.createServer(function (req, res){  
  //cuerpo del callback  
}).listen(3030, 'localhost');
```

CÓMO USAR HTTP

Este método recibe como parámetro un callback, que se ejecutará **cada vez** que se envíe un request al servidor. El callback recibirá dos parámetros: el primero representa los datos que envió el cliente como solicitud (`request`), el segundo representa la respuesta que le enviará el servidor al cliente (`response`).

```
{ http.createServer(function (req, res){  
  //cuerpo del callback  
});
```

3

4

QUÉ ES UN PUERTO

¿Qué pasaría si vamos a un edificio a visitar a alguien, pero no sabemos ni el piso, ni el departamento? En ese caso no sabríamos qué timbre tocar y quedaríamos a la espera sin poder hacer demasiado.

Nosotros necesitamos saber el timbre para poder, efectivamente, visitar a esa persona.

Lo mismo pasa con un **servidor**. Necesita saber el **puerto**, que no es más que un **número** que representa una **dirección específica** en donde irá a procesar las peticiones del cliente.

5

6

Hasta el momento:

Creamos un servidor que corre en el dominio *localhost* y escucha en el **puerto** 3030.



LEVANTAR EL SERVIDOR

Para levantar el servidor haremos uso de la consola para ejecutar nuestro archivo **entry-point**.

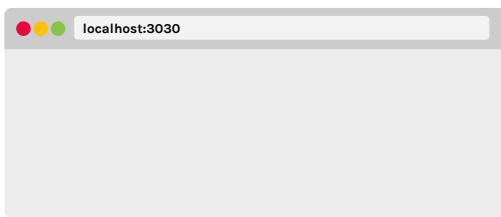
```
>_ node app.js
```

Al ejecutar ese comando, la consola quedará inhabilitada, sin poder escribir ningún comando sobre ella. Esto sucede porque, una vez levantado el servidor, el mismo se queda "escuchando" por los request y response en el puerto definido.

Para cortar el servidor, presionamos `ctrl + c` para windows y linux, `cmd + c` para mac.

TESTEAR EL SERVIDOR

Para testear, le pediremos al navegador que le haga un **request** al servidor, en el puerto y dominio que definimos.



Ya tenemos la estructura necesaria para hacer las **peticiones** al servidor.

Es momento de definir el **response** que nos dará el mismo.



DEFINIENDO EL RESPONSE

Lo primero que hay que hacer es definir las cabeceras.

Para crearlas usaremos el método `writeHead()` que lo ejecutaremos sobre el parámetro `res`, que será el **response**. El método recibe dos parámetros: el primero, un número de 4 dígitos que representará el **status** de la petición, el segundo, un objeto literal que define el tipo de contenido que se le está enviando al cliente.

```
http.createServer(function (req, res){  
  res.writeHead(200, {"Content-Type": "text/plain"});  
}).listen(3030, 'localhost');
```

DEFINIENDO EL RESPONSE

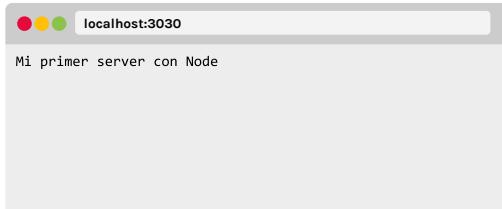
Ahora es momento de definir el contenido que le enviaremos al cliente. Para eso usaremos el método `end()` que recibe como parámetro un **string**, que representará el cuerpo del contenido que estaremos enviando.

Este método debe ir **siempre** después de la definición de las cabeceras, y cuando éste termina, **cierre el ciclo** del response.

```
http.createServer(function (req, res){  
  res.writeHead(200, {"Content-Type": "text/plain"});  
  res.end('Mi primer server con Node');  
}).listen(3030, 'localhost');
```

TESTEAR EL SERVIDOR

Al levantar el servidor y hacer el **request**, veremos en el navegador el string que definimos como **response**.



13

PROCESO ROUTING

Por cada url que escribamos en el navegador, estamos haciendo un **request** diferente, y por lo tanto, esperando una **respuesta** específica en cada caso. Se define **routing** al proceso en que definimos esas rutas y sus respuestas.

Dentro de la estructura que definimos para crear el servidor, contamos con el parámetro `req` que será el request que envíe el cliente. El mismo es un dato de tipo **objeto**, con propiedades y funcionalidades.

La propiedad `url` nos permite saber qué url ingresó el cliente al momento de hacer el **request**.

14

{ código }

```
http.createServer(function (req, res){  
  res.writeHead(200, {"Content-Type": "text/plain"});  
  if(req.url == '/'){  
    res.end('Mi primer server con Node');  
  }  
  if(req.url == '/saludo'){  
    res.end('Hola! Estamos en localhost:3030/saludo');  
  }  
}).listen(3030, 'localhost');
```

{ código }

```
http.createServer(function (req, res){  
  res.writeHead(200, {"Content-Type": "text/plain"});  
  if(req.url == '/'){  
    res.end('Mi primer server con Node');  
  }  
  if(req.url == '/saludo'){  
    res.end('Hola! Estamos en localhost:3030/saludo');  
  }  
}).listen(3030, 'localhost');
```

Definimos una estructura `if` y preguntamos si el valor que viene en la propiedad `url` del objeto `request` es `/`, es decir, si el cliente solicitó la ruta raíz de esta aplicación: `localhost:3030`.

15

16

{ código }

```
http.createServer(function (req, res){  
  res.writeHead(200, {"Content-Type": "text/plain"});  
  if(req.url == '/'){  
    res.end('Mi primer server con Node');  
  }  
  if(req.url == '/saludo'){  
    res.end('Hola! Estamos en localhost:3030/saludo');  
  }  
}).listen(3030, 'localhost');
```

{ código }

Si la condición es verdadera,
envío como **response** el string: Mi
primer server con Node.

```
http.createServer(function (req, res){  
  res.writeHead(200, {"Content-Type": "text/plain"});  
  if(req.url == '/'){  
    res.end('Mi primer server con Node');  
  }  
  if(req.url == '/saludo'){  
    res.end('Hola! Estamos en localhost:3030/saludo');  
  }  
}).listen(3030, 'localhost');
```

Defino otra estructura `if` y pregunto si la ruta que llegó por `request` es `/saludo`.

17

18

{ código }

```
http.createServer(function (req, res){  
  res.writeHead(200, {"Content-Type": "text/plain"});  
  if(req.url == '/'){  
    res.end('Mi primer server con Node');  
  }  
  if(req.url == '/saludo'){  
    res.end('Hola! Estamos en localhost:3030/saludo');  
  }  
}).listen(3030, 'localhost');
```

Si la condición es **verdadera**, envío como **response** el string: Hola!
Estamos en localhost:3030/saludo

La **cantidad** de rutas que definamos va a depender exclusivamente de la **aplicación** que estemos desarrollando y de los **response** que queramos dar.



INTRODUCCIÓN A EXPRESS

Es un **framework** que facilita y agiliza el **desarrollo** de aplicaciones web en **Node JS**.



QUÉ ES UN FRAMEWORK

Un framework es un **entorno de trabajo** que trae resueltas una serie de tareas, automatizando así el desarrollo de cualquier aplicación.

Los **frameworks** de **Node.js** se utilizan principalmente por su productividad, escalabilidad y velocidad. **Express** es uno de los más populares y estable y es muy utilizado tanto para aplicaciones web como para mobile.

Express

3

CÓMO USAR EXPRESS

Lo primero que hay que hacer es **instalar la librería** en un proyecto Node ya inicializado, es decir, haber hecho `npm init` y tener creado el archivo `package.json`.

```
>_ npm install express --save
```

Con el comando `--save` estamos **guardando**, en la propiedad `dependencies` del archivo `package.json`, una **referencia** a la **librería** que estamos instalando. De esta manera, quien quiera clonar el proyecto, podrá instalar todas las dependencias que el mismo necesita para funcionar haciendo uso de `npm install`.

CÓMO USAR EXPRESS

Una vez instalado Express, tendremos que **requerir** el **módulo** en nuestro entry-point, `app.js`.

```
{ const express = require('express');
```

Lo que devuelve la librería es una **función** que encapsula todas las funcionalidades de Express y para poder empezar a usarlas, hace falta **ejecutar** esa función. Lo próximo, entonces, sería crear una **variable nueva** y almacenar en ella la **ejecución** de `express` y así poder tener todos los métodos de la librería disponibles.

```
{ const app = express();
```

5

Cada vez que necesitemos trabajar con Express, hace falta instalarlo **dentro** de cada **proyecto** que estemos desarrollando.



“

RECORRIDA ECOSISTEMA DE EXPRESS

Dentro de este framework encontramos **funcionalidades** completamente **listas** para usar. Todas ellas **conviven** entre sí, creando juntas lo que llamamos el **ecosistema de Express**.



FUNCIÓN DE ALTO NIVEL

Express cuenta con una función de alto nivel que, al invocarla, retorna un **objeto** con múltiples propiedades y métodos: la función `express()`. Al **almacenar** la ejecución de esa función en una constante, podremos tener **acceso** a través de ella a todas esas funcionalidades que nos da el objeto.

`const app = express();`

Constante
Aquí tendremos almacenado el **objeto**. Usando la notación de punto podremos acceder a todas sus **propiedades** y **métodos**.

Función alto nivel
La función **ejecutada** devuelve un **objeto**.

3

FUNCIONALIDADES EXPRESS

El objeto que devuelve `express()` tiene métodos asociados para gestionar las peticiones que se hagan a través de get, post, delete, put y patch.

```
app.get();
app.post();
app.put();
app.patch();
app.delete();
```

Cuando trabajemos con alguno de esos métodos, tendremos acceso a dos **objetos literales** que nos da Express: **request** y **response**, que nos permitirán acceder a los **requests del cliente** y a los **responses del servidor**.



4

SERVIDOR HTTP EN EXPRESS

Una de las tantas **funcionalidades** que trae lista Express es la posibilidad de **levantar un servidor** de manera muy **sencilla** y en pocos pasos.



{ código }

```
const express = require('express');
```

Requerimos el módulo de **Express** y almacenamos la función que nos devuelve en la constante **express**.

{ código }

```
const express = require('express');
const app = express();
```

Ejecutamos la función y almacenamos el objeto que devuelve en la constante **app**.

Ahora, a través de la misma vamos a tener acceso a todas las propiedades y métodos que nos da Express.

3

4

{ código }

```
const express = require('express');
const app = express();
```

```
app.listen();
```

Al objeto **app** le pedimos el método **listen**, que se encargará de levantar el servidor.

{ código }

```
const express = require('express');
const app = express();
```

```
app.listen(3000, () => console.log('Servidor corriendo'));
```

El método recibe dos parámetros:

El primero, el número de puerto en el que queremos que se ejecute la aplicación.

El segundo (opcional), un callback que retorna un `console.log` para saber si el servidor se levantó correctamente.

5

6

Con el servidor levantado sólo nos faltaría **definir** las **rutas** para empezar a manejar los **response** de nuestra aplicación.



{ código }

```
const express = require('express');
const app = express();

app.listen(3000, () => console.log('Servidor corriendo'));

app.get('/', (req, res) => {
    res.send('¡Hola mundo!');
})
```

Al objeto **app** le pedimos el método **get**.

8

{ código }

```
const express = require('express');
const app = express();

app.listen(3000, () => console.log('Servidor corriendo'))

app.get('/', (req, res) => {
    res.send('¡Hola mundo!');
})
```

El método recibe dos parámetros:
El primero, un string que define la url de la ruta.
El segundo, un callback con dos parámetros: objetos **request** y **response** que nos pone a disposición Express cada vez que trabajamos con algún método de petición HTTP.

{ código }

```
const express = require('express');
const app = express();

app.listen(3000, () => console.log('Servidor corriendo'));

app.get('/', (req, res) => {
    res.send('¡Hola mundo!');
})
```

El nombre de esos parámetros puede ser el que queramos. Se está usando **req** para **request** y **res** para **response**.

9

10

{ código }

```
const express = require('express');
const app = express();

app.listen(3000, () => console.log('Servidor corriendo'))

app.get('/', (req, res) => {
    res.send('¡Hola mundo!');
})
```

Dentro del callback definimos la respuesta que enviaremos.
Al objeto **res (response)**, le pedimos el método **send**. Como parámetro le pasamos lo que queremos mostrar en el browser. En este caso, el texto ¡Hola mundo!.

11

“

RUTAS EN EXPRESS

A través del **sistema de ruteo** de Express podemos definir, de manera sencilla, cómo va a responder nuestra aplicación según el **método HTTP** y la **ruta** que esté llegando al **servidor**.



DEFINIENDO UNA RUTA

A través de Express, contamos con una **estructura básica** para definir cada una de las rutas y sus respuestas de nuestra aplicación:

```
app.get('/', function (req , res) {
  res.send('¡Hola mundo!');
})
```

3

DEFINIENDO UNA RUTA

A través de Express, contamos con una **estructura básica** para definir cada una de las rutas y sus respuestas de nuestra aplicación:

```
app.get('/', function (req , res) {
  res.send('¡Hola mundo!');
})
```

Método

Escribimos el **método HTTP** que queremos atender:
get, post, put, patch ó delete.

5

DEFINIENDO UNA RUTA

A través de Express, contamos con una **estructura básica** para definir cada una de las rutas y sus respuestas de nuestra aplicación:

```
app.get('/', function (req , res) {
  res.send('¡Hola mundo!');
})
```

Variable que guarda la **ejecución** de Express.

4

DEFINIENDO UNA RUTA

A través de Express, contamos con una **estructura básica** para definir cada una de las rutas y sus respuestas de nuestra aplicación:

```
app.get('/', function (req , res) {
  res.send('¡Hola mundo!');
})
```

Path

String que hará referencia a la **ruta** en sí (url que llegará por petición).

6

DEFINIENDO UNA RUTA

A través de Express, contamos con una **estructura básica** para definir cada una de las rutas y sus respuestas de nuestra aplicación:

```
app.get('/', function (req , res) {  
    res.send('¡Hola mundo!');  
})
```

Handler

Callback que se encargará de definir qué acción tomar cuando se acceda a la ruta definida.

DEFINIENDO UNA RUTA

A través de Express, contamos con una **estructura básica** para definir cada una de las rutas y sus respuestas de nuestra aplicación:

```
app.get('/', function (req , res) {  
    res.send('¡Hola mundo!');  
})
```

Request (primer parámetro del handler)

Es un objeto literal con múltiples métodos y propiedades. Representa al **request** solicitado.

7

DEFINIENDO UNA RUTA

A través de Express, contamos con una **estructura básica** para definir cada una de las rutas y sus respuestas de nuestra aplicación:

```
app.get('/', function (req , res) {  
    res.send('¡Hola mundo!');  
})
```

Response (segundo parámetro del handler)

Es un objeto literal con múltiples métodos y propiedades. Representa al **response** que dará el servidor.

DEFINIENDO UNA RUTA

A través de Express, contamos con una **estructura básica** para definir cada una de las rutas y sus respuestas de nuestra aplicación:

```
app.get('/', function (req , res) {  
    res.send('¡Hola mundo!');  
})
```

Lógica de la ruta

Definimos la **lógica** que va a manejar la ruta definida. Se estila dar la respuesta que verá el cliente en su navegador.

9

8

10



Módulo: HTML y CSS

ETIQUETAS, ATRIBUTOS Y ESTRUCTURA BÁSICA



¿CÓMO VEMOS NORMALMENTE UNA PÁGINA WEB?

HTML

Hyper Text Markup Language

Lenguaje de marcado de hipertexto.

Compuesto de ELEMENTOS formados por ETIQUETAS y ATRIBUTOS



¿Ciencia de cohetes?



¡No! ¡Es mucho más fácil de lo que parece!

1. ADN DE UNA PÁGINA WEB



¿Y QUÉ HAY DETRÁS DE LO QUE VEMOS EN EL NAVEGADOR?

2. SINTAXIS DE UN ELEMENTO

SINTAXIS DE UN ELEMENTO

<h1> ... </h1>

Etiqueta de apertura

Indica el comienzo de un elemento, siempre debe iniciar con el símbolo de menor < y terminar con el signo de mayor >.

Dentro debe ir el **nombre** de la etiqueta.

Etiqueta de clausura

Indica el final de un elemento, siempre debe iniciar con el símbolo de menor seguido de la barra diagonal </ y terminar con el signo de mayor >.

El **nombre** va adentro.

7

SINTAXIS DE UN ELEMENTO

<h1 align="center"> ... </h1>

Atributos

Son configuraciones adicionales de los elementos que ajustan su comportamiento de diversas formas.

Valores

Nos permiten definir las configuraciones. Siempre van a estar escritos entre comillas "" y luego de un signo de igual =.

Contenido

Todo aquello que escribamos entre las etiquetas de apertura y cierre de un elemento, conformarán su contenido.

8

REPASEMOS:

ELEMENTO

<h1 align="center">	APERTURA
Hola Mundo	CONTENIDO
</h1>	CIERRE

¿Cómo se compone un elemento?

1. Una etiqueta de **apertura**
 - a. Opcionalmente uno o más **atributos**
2. El **contenido**
3. Una etiqueta de **cierre**. Algunos elementos no la llevan.

9

3.

ESTRUCTURA BÁSICA DE UN DOCUMENTO HTML

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Hello world</title>
  </head>
  <body>
    <p>Lorem ipsum dolor sit amet,
    consectetur adipisicing elit.
    Tenetur deserunt molestiae
    numquam veritatis ea ut
    praesentium explicabo atque
    maxime a eaque, aut id
    consequuntur. Et nemo non
    perspiciatis eum!</p>
  </body>
</html>
```



<!DOCTYPE html>	VERSIÓN DE HTML
<html>	INICIO DE LA PÁGINA
<head>	
<meta charset="utf-8">	
<title>Hello world</title>	CONFIGURACIÓN DE LA PÁGINA
</head>	
<body>	
<p>Lorem ipsum dolor sit amet,	CONTENIDO DE LA PÁGINA
consectetur adipisicing elit.	
Tenetur deserunt molestiae	
numquam veritatis ea ut	
praesentium explicabo atque	
maxime a eaque, aut id	
consequuntur. Et nemo non	
perspiciatis eum!</p>	
</body>	
</html>	FIN DE LA PÁGINA

10

8



Módulo: HTML y CSS

CONFIGURACIÓN Y ETIQUETAS DE TEXTO

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">           CONFIGURACIÓN DE LA PÁGINA
    <title>Hello world</title>
  </head>
  <body>
    <p>Lorem ipsum dolor sit amet,
    consectetur adipisicing elit.
    Tenetur deserunt molestiae
    numquam veritatis ea ut
    praesentium explicabo atque
    maxime a eaque, aut id
    consequuntur. Et nemo non
    perspiciatis eum!</p>
  </body>
</html>
```

3

¿Ciencia de cohetes?



iNo! iEs mucho más fácil de lo que parece!

1.

CONFIGURACIÓN DE CARACTERES

EL META CHARSET Y SU USO

<meta charset="utf-8">

Etiqueta → meta

Permite definir propiedades de la página que no pueden definirse en otras etiquetas como <title>, <link> o <style>. Suelen ser datos sobre el contenido, su descripción, autor, caracteres, etc.

Atributo → charset

Permite definir la codificación de caracteres a utilizar. HTML funciona por defecto para el inglés, cuando queremos utilizar caracteres de otros idiomas, debemos especificarlos.

Valor → utf-8

UTF8 es la codificación de caracteres más utilizada en HTML. Nos permite por ejemplo mostrar correctamente la eñe y los acentos del español. iSoporta más de 60 idiomas!

Este es mi título

Éste es un párrafo que tiene muchos acentos... á, é, í, ó, ú. ¿Se verán bien? ¿Se verán caracteres extraños?

Éste es otro párrafo de prueba. El veloz murciélagos hindú comía feliz cardillo y kiwi. La cigüeña toca el saxofón detrás del palenque de paja.

¡Un párrafo más! El veloz murciélagos hindú comía feliz cardillo y kiwi. La cigüeña toca el saxofón detrás del palenque de paja.

EJEMPLO DE
HTML CON
META CHARSET

UTF-8

5



EJEMPLO DE
HTML SIN
META CHARSET

UTF-8

Á%oeste es mi tÃtulo

Á%oeste es un pÃjrrafo que tiene muchos acentos... Ái, Á©, Á, Á³, Áº. ¿Se verÃjn bien? ¿Se verÃjn caracteres extraÃ±os?

Á%oeste es otro pÃjrrafo de prueba. El veloz murciÃ©lago hindÃº comÃa feliz cardillo y kiwi. La cigÃ½eÃ±a toca el saxofÃ³n detrÃjs del palenque de paja.

ÁjUn pÃjrrafo mÃjs! El veloz murciÃ©lago hindÃº comÃa feliz cardillo y kiwi. La cigÃ½eÃ±a toca el saxofÃ³n detrÃjs del palenque de paja.

6

2.

ETIQUETAS DE TEXTO

...



ELEMENTOS DE ENCABEZADO

Los elementos de encabezado implementan seis niveles de encabezado del documento, `<h1>` es el más importante, y `<h6>`, el menos importante. Un elemento de encabezado describe brevemente el tema de la sección que presenta.

```
<h1> Título Principal </h1>
<h2> Título Secundario </h2>
<h3> Título Principal </h3>
<h4> Título Principal </h4>
<h5> Título Principal </h5>
<h6> Título Principal </h6>
```

8

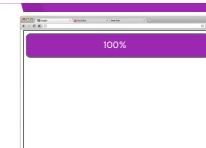
ELEMENTOS DE ENCABEZADO



El elemento `<h1>`, por recomendación de la **W3C**, sólo debe ser utilizado **una vez** por documento HTML.

```
<h1> Título Principal </h1>
<h1> Otro Título </h1> ❌
<h2> Otro Título </h2> ✓
```

9



ELEMENTOS DE PÁRRAFO

Los elementos de párrafo `<p>`, nos permiten distribuir el texto en párrafos. Podemos usar tantos como necesitemos.

```
<p>Éste es un párrafo, puede tener todo el  
texto que necesitemos.</p>
```

```
<p>El navegador se encargará de agregar  
espacio vertical entre cada uno de los  
párrafos que escribamos.</p>
```

10



Módulo: HTML y CSS

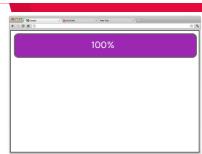
LISTAS

1.

LISTAS ORDENADAS

LISTAS ORDENADAS

Las listas ordenadas nos permiten enumerar ítems de manera consecutiva. Por defecto van a empezar en el número 1 y se irán incrementando con cada ítem nuevo.

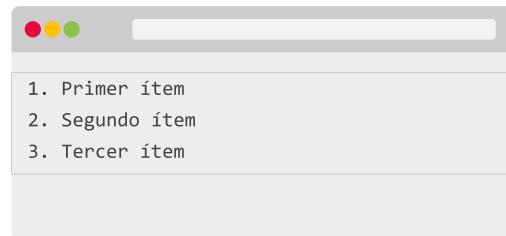


```
<ol>
  <li>Primer ítem</li>
  <li>Segundo ítem</li>
  <li>Tercer ítem</li>
</ol>
```

3

LISTAS ORDENADAS

Las listas ordenadas se van a ver de la siguiente manera en el navegador:



EDITANDO NUESTRAS LISTAS ORDENADAS

```
<ol type="1">
```

Atributo → type	Valor
Nos permite cambiar el tipo de viñeta de la lista.	1 Numérica
	A Alfabética
	I Numérica romana

5

EDITANDO NUESTRAS LISTAS ORDENADAS

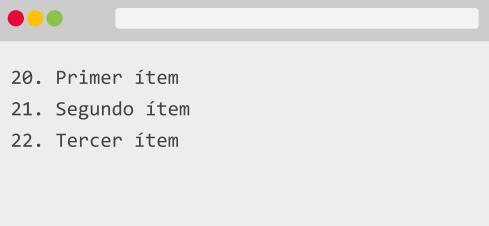
```
<ol start="20">
```

Atributo → start	Valor
Nos permite definir dónde va a empezar nuestra numeración	Puede ser cualquier número positivo o negativo.

6

LISTAS ORDENADAS

Usando el parámetro start, modificamos dónde empieza nuestra lista.

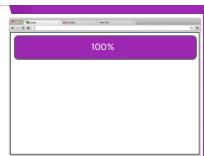


2.

LISTAS DESORDENADAS

LISTAS DESORDENADAS

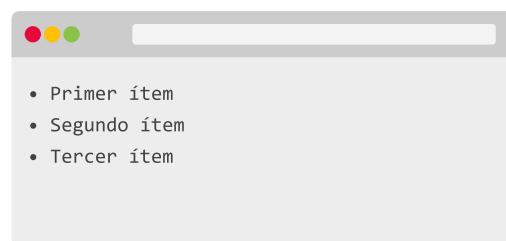
Las listas desordenadas también nos permiten listar ítems. Por defecto va a generar una viñeta tipo “bolita” por cada ítem nuevo que se agregue.



```
<ul>
  <li>Primer ítem</li>
  <li>Segundo ítem</li>
  <li>Tercer ítem</li>
</ul>
```

LISTAS DESORDENADAS

Las listas desordenadas se van a ver de la siguiente manera en el navegador:



EDITANDO NUESTRAS LISTAS DESORDENADAS

```
<ol type="bullet">
```

Atributo → type

Nos permite cambiar el tipo de viñeta de la lista.

Valor

circle ○
square ■
upper-roman I
lower-alpha a

3.

LISTAS ANIDADAS

LISTAS ANIDADAS

Las listas anidadas nos permiten crear varios niveles de jerarquía y organización. Las podemos anidar como deseemos y generar los niveles que necesitemos.

```
<ul>
  <li>
    Recordar para el viaje:
    <ol>
      <li>Dni</li>
      <li>Pasajes</li>
    </ol>
  </li>
  <li>Llamar a Assist Card</li>
  <li>Adaptador para celular</li>
</ul>
```

IMPORTANTE



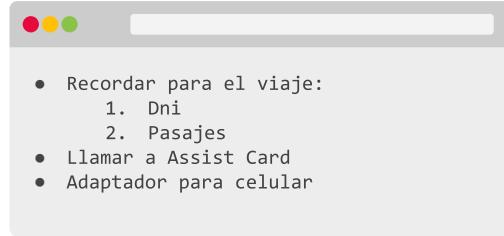
Dentro de un `` o un `` sólo pueden haber elementos ``.

Las listas se pueden anidar agregando un `` u `` dentro de un ``.

13

LISTAS ANIDADAS EN EL NAVEGADOR

Así se verán nuestras listas:



14



Módulo: HTML y CSS

RUTAS: HIPERVÍNCULOS E IMÁGENES

QUÉ ES UNA RUTA

Es una dirección o camino (también conocido con el término inglés 'path'), que le va a permitir al navegador encontrar un recurso. Ese recurso puede ser otra página web, una imagen, un video o cualquier otro tipo de archivo.

En el caso de los **enlaces**, la ruta indica la dirección a la que tiene que llevarnos el navegador cuando pulsamos sobre él.

Nos podemos encontrar dos tipos de rutas distintas:

RUTA ABSOLUTA

Puedo acceder a ese recurso desde cualquier sitio donde esté.

Ejemplo:

<https://www.google.com/>

RUTA RELATIVA

Va a depender de dónde esté en ese momento para poder acceder al recurso. Es decir, es relativa a mi posición actual.

Ejemplo:

[..//imágenes/fotoPerfil.jpg](#)

3

EJEMPLO RUTA RELATIVA



Una imagen que está alojada en una carpeta determinada. Su ruta, es decir, el acceso hacia ese recurso, va a variar según el lugar desde el que quiera acceder al mismo.

5

1. RUTAS

EJEMPLO RUTA ABSOLUTA



Una imagen que está alojada en una url. Puedo acceder a ella sin importar el lugar en el que esté navegando en ese momento. Su ruta, es decir, el acceso hacia ese recurso, es siempre el mismo.



home.html



En este caso, como el archivo **home.html** y **foto.jpg** se encuentran en la misma carpeta basta con simplemente escribir el nombre del archivo para referenciarlo.

La ruta relativa sería **foto.jpg**

4

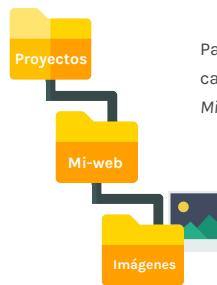
EJEMPLO RUTA RELATIVA



Para crear una ruta hacia la **imagen** desde la carpeta **Mi-web**, debería quedar así:
`Imagenes/foto.jpg`

7

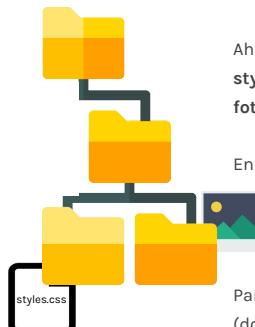
EJEMPLO RUTA RELATIVA



Para crear una ruta hacia la **imagen** desde la carpeta **Proyectos**, debería quedar así:
`Mi-web/Imagenes/foto.jpg`

8

EJEMPLO RUTA RELATIVA



Ahora... ¿Qué pasaría si estuviese en **styles.css** y quisiera hacer referencia a **foto.jpg**?

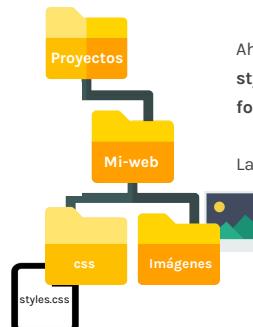
En este caso tendríamos que:

- Salir un nivel
- Entrar en la carpeta **Imagenes**
- Referenciar **foto.jpg**

Para salir un nivel se utilizan los caracteres ..
(dos puntos, uno al lado del otro)

9

EJEMPLO RUTA RELATIVA



Ahora... ¿Qué pasaría si estuviese en **styles.css** y quisiera hacer referencia a **foto.jpg**?

La solución sería:

`..../Imagenes/foto.jpg`

10

2.

HIPERVÍNCULOS O ENLACES

11

ENLACES

A través de la etiqueta `<a>` vamos a poder crear nuestros enlaces. Ésta es una etiqueta de apertura y cierre: `<a>`.

`
| ¡Vamos a Google!
`

Este **atributo** se usa para indicar el destino al que apunta nuestro enlace.

Aquí podremos escribir el texto que verá el usuario. También podemos poner otros elementos de HTML como imágenes.



Aquí podremos especificar una ruta absoluta (como el ejemplo) o una relativa.

12

TIPOS DE ENLACES

EXTERNOS

Sus rutas están fuera de nuestro sitio, son siempre absolutas.

```
<a href="https://www.youtube.com/">Ir a youtube</a>
```

LOCALES

Sus rutas están dentro de nuestro sitio, se recomienda que sean relativas siempre que sea posible.

```
<a href="inicio.html">Inicio</a>
```

ANCLAS

Sirven para hacer referencia a una determinada parte de una página, como puede ser una sección o un titular. Inician con el carácter #.

```
<a href="#biografia">Biografía</a>
```

13

TIPOS DE ENLACES

ANCLAS (continuación)

Las anclas pueden combinarse con todas las anteriores.

```
<a href="https://www.sitio.com/#contacto">Contactanos</a>
```

```
<a href="sobre-nosotros.html#elEquipo">Nuestro equipo</a>
```

CORREO

Al hacer clic en ellos, se abrirá el programa de correo que tengamos como predeterminado para enviar un correo a esa dirección de email.

```
<a href="mailto:user@server.com">Dejanos tu mensaje</a>
```

TELÉFONO

Parecido al caso anterior, si estamos usando nuestro smartphone, se iniciará una llamada a ese número o aparecerá listo para llamar.

```
<a href="tel:1145678900">Llamanos!</a>
```

14

3. IMÁGENES

TEXTO ALTERNATIVO

El texto alternativo nos permite describir una imagen. Se muestra cuando la imagen no carga por alguna razón o para las personas que usen lectores de pantalla (ej: no videntes).

```

```

Este atributo nos permite especificar el texto alternativo.

Hasta 125 caracteres, debe contener una descripción corta de la imagen de deberíamos estar viendo.

SABÍAS QUE...

El atributo alt también sirve para que los buscadores puedan entender nuestras imágenes y mejora el posicionamiento de nuestro sitio en las búsquedas.

17



FUENTE DE UNA IMAGEN

Dentro de nuestro documento html podemos agregar imágenes a través de la etiqueta . Esta etiqueta nos permite **invocar** las imágenes, es decir, hacer referencia al lugar donde están alojadas para que aparezcan en el navegador.

```

```

src

Este atributo se usa para indicar el destino en donde está alojada nuestra imagen.

Ruta

Aquí podremos especificar una ruta absoluta como el ejemplo o una relativa.

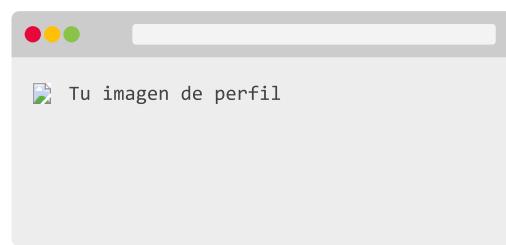
IMPORTANTE ⚠

Las imágenes no llevan etiqueta de cierre ().

16

TEXTO ALTERNATIVO

Así verá el usuario nuestra imagen en caso de que no cargue. Según qué navegador esté usando, puede verse distinto.



18

ANCHOS Y ALTOS

```
<img width="320" height="150">  
<img width="50%" height="50%">
```

width

Nos permite indicar el **ancho** de nuestra imagen. (*No es obligatorio*).

Los **valores** pueden ser tanto en píxeles (sólo escribimos el número) como en porcentajes (con el % al final).

height

Nos permite indicar el **alto** de nuestra imagen. (*No es obligatorio*).

Los **valores** pueden ser tanto en píxeles (sólo escribimos el número) como en porcentajes (con el % al final).



Módulo: HTML y CSS

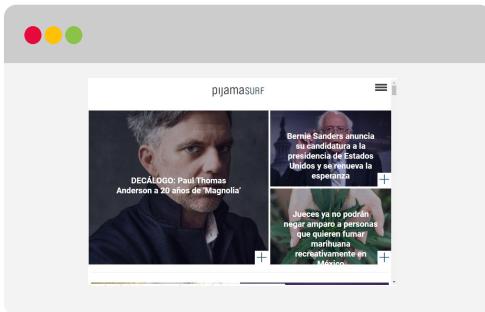
ETIQUETAS MULTIMEDIA

ETIQUETA IFRADE

A través de esta etiqueta vamos a poder insertar contenido de otros sitios webs **dentro** de nuestro sitio. Este contenido va a traer incorporado estilos y funcionalidades del sitio al que pertenece. Usando los atributos **width** y **height**, le podemos dar tamaño.

```
<iframe src="https://www.ole.com.ar/"></iframe>
```

ETIQUETA IFRADE



3

ETIQUETA VIDEO

```
<video width="480" poster="video.jpg" controls>
  <source src="video.mp4" type="video/mp4">
  <source src="video.ogg" type="video/ogg">
  <source src="video.webm" type="video/webm">
  <p>Tu navegador no soporta video html5</p>
</video>
```

ETIQUETA VIDEO



5

ETIQUETA AUDIO

```
<audio width="400" controls>
  <source src="audio.mp3" type="audio/mp3">
  <source src="audio.wav" type="audio/x-wav">
  <source src="audio.ogg" type="audio/ogg">
  <p>Tu navegador no soporta audio html5</p>
</audio>
```

2

4

6

ETIQUETA AUDIO



7



Módulo: HTML y CSS

INTRODUCCIÓN A CSS

CSS

Cascading Style Sheets

Hojas de estilo en cascada.

Compuestas de REGLAS,
SELECTORES y
DECLARACIONES.

1. ¿QUÉ SIGNIFICA CSS?

```

192 .mfp-arrow-a::before, .mfp-arrow-a::after {
193   border-top-width: 13px;
194   border-bottom-width: 13px;
195   top: 0px;
196   border-top-width: 21px;
197   border-bottom-width: 21px;
198   opacity: 0.7;
199 }
200 .mfp-arrow-left {
201   left: 0;
202   .mfp-arrow-left::after, .mfp-arrow-left::before, .mfp-a {
203     margin-left: 31px;
204     .mfp-arrow-left::before, .mfp-arrow-left::after, .mfp-b {
205     margin-left: 25px;
206     border-right: 27px solid #3F3F3F;
207   }
208   .mfp-arrow-right {
209     right: 0;
210   }
211   .mfp-arrow-right::after, .mfp-arrow-right::before, .mfp-a {
212     border-left: 17px solid #3F3F3F;
213     margin-left: 31px;
214     .mfp-arrow-right::before, .mfp-arrow-right::after, .mfp-b {
215     border-left: 27px solid #3F3F3F;
216   }
217   padding-top: 40px;
218   padding-bottom: 40px;

```

¿Y PARA QUÉ NOS SIRVEN LAS HOJAS DE ESTILO?

¡DEFINAMOS!

Las hojas de estilo sirven para **estilizar** nuestro contenido HTML. Con **CSS** podemos cambiar colores, fondos, tipografías, anchos, altos, etc. Así como también generar animaciones y transiciones.

Contamos con **3 métodos** para vincular nuestros archivos CSS con el documento HTML:

VINCULACIÓN INTERNA

A través de la etiqueta **<style>** dentro del **<head>**.

VINCULACIÓN EN LÍNEA

Usando el atributo **style** en cada elemento de nuestro HTML.

`<p style="color:red"></p>`

Y LA MÁS USADA...

VINCULACIÓN EXTERNA

Escribiendo todos nuestros estilos en un archivo CSS y vinculándolos, usando la etiqueta **<link>** dentro del **<head>** de nuestro documento.

`<link href="css/estilos.css" rel="stylesheet">`

valor

Ruta de la ubicación de mi hoja de estilo

rel

Indica qué relación hay entre los documentos a enlazar

valor

Este valor siempre es el mismo.

2.

SELECTORES DE CSS

Los **selectores** nos van a permitir "atrapar" a los elementos HTML que queramos modificar y así aplicarles nuestros estilos.

SELECTORES DE CSS

COMBINADOS

Afectan a los que cumplan todas las condiciones.

Ejemplo: `p.saludo` (estamos seleccionando los `<p>` que tengan asignada la clase `saludo`)

DESCENDENTES

Sirven para agregar especificidad. Estos se utilizan con un espacio.

Ejemplo: `ul li.especial` (estamos seleccionando los `` dentro de un `` que tengan asignada la clase `especial`)

SELECTORES DE CSS

Los **selectores** nos van a permitir "atrapar" a los elementos HTML que queramos modificar y así aplicarles nuestros estilos.

ID `<p id="saludo"></p>`

A través del atributo `id` le asignamos un nombre al elemento.

Sintaxis: `#saludo` (se usa el `#` seguido del nombre)

CLASE `<h1 class="títulos"></h1>`

A través del atributo `class` le asignamos un nombre al elemento.

Sintaxis: `.títulos` (se usa el `.` seguido del nombre)

ETIQUETA ``

Afecta a la etiqueta que nombremos en nuestro CSS.

Sintaxis: `ul` (se usa el nombre de la etiqueta)

9

SELECTORES DE CLASE

Este selector va a atrapar al elemento HTML que tenga asignado el **atributo CLASS**. Podemos asignarle la cantidad de clases que queramos a un mismo elemento. Para hacerlo, sólo hace falta separarlas con un espacio.

`<h1 class="titulo noticias">Noticias</h1>`

SINTAXIS

`.titulo`

Para llamarlo desde el css usamos el `.` seguido del **nombre de la CLASE**.

EJEMPLO

```
.titulo {  
    font-size: 22px;  
}
```

11

SELECTORES DE ETIQUETA

Este selector va a atrapar al elemento HTML con el mismo nombre de etiqueta que llamemos desde nuestro CSS.

`<p>Párrafo uno</p>`

SINTAXIS

`p`

Para llamarlo desde el css usamos el **nombre de la ETIQUETA**.

EJEMPLO

```
p {  
    color: gray;  
}
```

8

10

12

SELECTORES COMBINADOS

Estos selectores afectan a el/los elemento/s que cumplan con todas las condiciones que establezca. En el siguiente ejemplo, vamos a atrapar al elemento **h2** que tenga asignada la clase **subtitulo**.

```
<h2 class="subtitulo">Subtítulo</h2>
```

SINTAXIS

h2.subtitulo

Para llamarlos desde el css sólo hace falta **agregar un selector al lado del otro**, cada uno con la sintaxis que le corresponda.

EJEMPLO

```
h2.subtitulo {  
    color: gray;  
}
```

13

SELECTORES DESCENDENTES

Estos selectores sirven para agregar especificidad. En el ejemplo vamos a atrapar al elemento **li** que esté dentro del **ul** con el id **lista**.

```
<ul id="lista">  
    <li>Primer ítem</li>  
</ul>
```

SINTAXIS

ul#saludo li

Para llamarlos desde el css escribimos los **selectores separados por un espacio** (esto indica parentesco)

EJEMPLO

```
ul#saludo li {  
    text-align: center;  
}
```

14

DATO IMPORTANTE



El CSS siempre va a priorizar a los selectores **más específicos** para aplicar los estilos.

15



Módulo: HTML y CSS

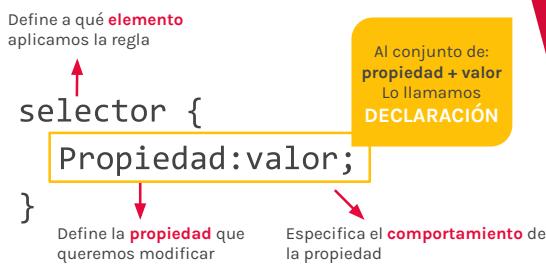
PROPIEDADES Y REGLAS CSS

1.

SINTAXIS CSS

QUÉ SON LAS REGLAS

Una regla de CSS es un conjunto de órdenes que se aplican a un elemento determinado para estilizar el mismo.



3

QUÉ SON LAS REGLAS

Así se vería una regla en nuestro css para modificar el estilo del body.

```
body {  
    background-color: purple;  
    font-family: Times New Roman;  
    text-align: center;  
}
```

4

2.

PROPIEDADES DE CSS

...

PROPIEDADES DE CSS

Existen muchas propiedades de CSS, las cuales nos permiten manipular los elementos del HTML a nuestro antojo. Si tuviéramos que agrupar dichas propiedades, los grupos serían más o menos así:

- Tipografías
- Visualización
- Fondos
- Comportamiento
- Tamaños
- Interfaz
- Posicionamiento
- Otros

6

PROPIEDADES PARA TIPOGRAFÍAS

font-family

Permite elegir la **familia tipográfica** que queremos usar.

font-size

Permite definir el **tamaño tipográfico**. Recibe un valor numérico acompañado de la unidad de medida. Las unidades de medida más habituales suelen ser: px, em, rem.

font-style

Define el **estilo de la tipografía**. El valor italic definiría una tipografía en cursiva. Para algunos tags el valor por default será italic.

7

PROPIEDADES PARA TIPOGRAFÍAS

font-weight

Define el **peso de la tipografía**. El valor bold definiría una tipografía en negrita. Para algunos tags el valor por default será bold.

text-align

Permite definir la **alineación del texto**. El valor por default para todos los tags es left.

text-decoration

Permite elegir un **tipo de decoración para el texto**. Para algunos tags el valor por default será underline.

8

PROPIEDADES PARA TIPOGRAFÍAS

font-family

Permite elegir la **familia tipográfica** que queremos usar. Como valor recibe el nombre de la tipografía que queramos usar.

```
p {  
    font-family: Arial;  
}
```

9

PROPIEDADES PARA TIPOGRAFÍAS

font-size

Permite definir el **tamaño tipográfico**. Recibe un valor numérico acompañado de la unidad de medida. Las unidades de medida más habituales suelen ser: px, em, rem.

```
p {  
    font-size: 23px;  
}
```

10

PROPIEDADES PARA TIPOGRAFÍAS

font-style

Define el **estilo de la tipografía**. Recibe los valores italic, normal y oblique. Para algunos tags el valor por default será italic.

```
p {  
    font-style: normal;  
}
```

11

PROPIEDADES PARA TIPOGRAFÍAS

font-weight

Define el **peso de la tipografía**. Recibe los valores bold, lighter, normal, entre otros. También puede recibir un valor numérico que se irá incrementando de 100 en 100. Para algunos tags el valor por default será bold.

```
p {  
    font-weight: normal;  
}           p {  
    font-weight: 200;  
}
```

12

PROPIEDADES PARA TIPOGRAFÍAS

text-align

Permite definir la **alineación** del **texto**. Los valores que recibe son center, left, right, inherit y justify. El valor por default para todos los tags es left.

```
p {  
    text-align: justify;  
}
```

13

PROPIEDADES PARA TIPOGRAFÍAS

text-decoration

Permite elegir un tipo de **decoración** para el **texto**. Recibe los valores line-through, underline, overline y none. Para algunos tags el valor por default será underline.

```
p {  
    text-decoration: underline;  
}
```

14

PROPIEDADES PARA TIPOGRAFÍAS

line-height

Permite definir el **interlineado** de los **textos**. Recibe un valor numérico acompañado de la unidad de medida. Tiene una relación directa con el font-size.

```
p {  
    line-height: 20px;  
}
```

15

PROPIEDADES PARA TIPOGRAFÍAS

Recordá que mencionamos al menos **8 grupos** de propiedades para CSS. Te dejamos un link para invitarte a investigar qué otras propiedades existen.

[CLICKEAME](#)

16



Módulo: HTML y CSS

COLORES Y FONDOS

FORMATOS DE COLOR

Los siguientes formatos se pueden aplicar en cualquier propiedad de CSS que reciba color.

HEXADECIMAL

#f05331

RGB

rgb(240, 83, 22)

[IMIRÁ ESTE SITIO!](#)

RGBA

rgba(240, 83, 22, 0.3)

El último número representa el nivel de opacidad que se le dará al elemento. Va del 0 al 1.

3

BACKGROUND-COLOR

Atributo que me permite asignarle un color de fondo a un elemento. Recibe como valor el nombre del color en inglés, así como también cualquiera de los formatos de color permitidos.

```
p {  
background-color: rgb(12, 34, 32);  
}
```

5

1.

REGLAS Y FORMATOS DE COLOR

COLOR

Atributo que me permite asignarle un color al texto de un elemento. Recibe como valor el nombre del color en inglés, así como también cualquiera de los formatos de color permitidos.

```
h2 {  
color: purple;  
}  
  
h2 {  
color: #345986;  
}
```

4

OPACIDAD

Mediante este atributo le otorgamos transparencia a todo el elemento, no solo al fondo.

opacity: 0.5;



El valor **red** representa el porcentaje de opacidad que le queremos dar al elemento. En este caso el 50%. No hay que escribir el símbolo de %.

0.2 = 20% de transparencia
0.05 = 5% de transparencia

6

2.

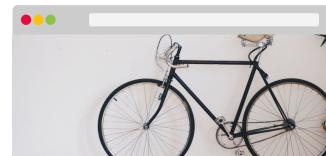
FONDOS DE IMAGEN

CSS define **6 propiedades** para establecer el fondo de cualquier elemento

BACKGROUND-IMAGE

Me permite asignarle una imagen de fondo al elemento, definiendo la ruta a través de la url.

```
body {  
    background-image: url('../img/bici.jpg');  
}
```

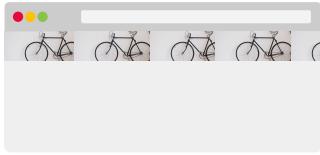


8

BACKGROUND-REPEAT

Me permite controlar si se va a repetir y de qué manera la imagen dispuesta. Recibe los valores repeat, no repeat, repeat-x, repeat-y, round y space.

```
body {  
    background-repeat: repeat-x;  
}
```

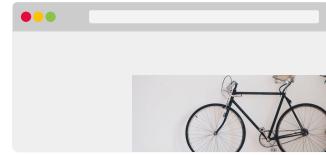


9

BACKGROUND-POSITION

Me permite mover la imagen dentro del elemento y decidir dónde colocarla. Recibe como valores tamaños en pixeles y porcentajes, así como también right, bottom, left, etc. Puedo asignarle uno o dos valores. El primero para especificar la posición en el eje x y el segundo la posición en el eje y.

```
body {  
    background-position: right bottom;  
}
```

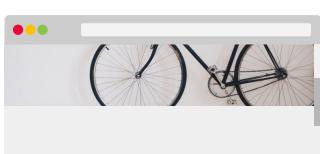


10

BACKGROUND-ATTACHMENT

Me permite establecer si la imagen de fondo se va a mover junto con la página al hacer scroll o si se va a quedar fija. Recibe como valor fixed, scroll, inherit y initial.

```
body {  
    background-attachment: fixed;  
}
```

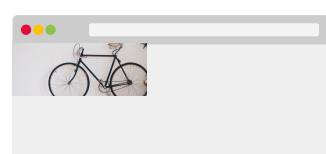


11

BACKGROUND-SIZE

Me permite establecer el tamaño de la imagen de fondo. Recibe como valor contain, cover, inherit, así como también tamaños en pixeles y porcentajes, indicando con el primer valor el ancho, y con el segundo el alto.

```
body {  
    background-size: 130px;  
}
```



12

BACKGROUND-COLOR

Me permite establecer un color al fondo del elemento. Recibe como valor el nombre del color en inglés, así como también cualquiera de los formatos de color permitidos.

```
body {  
    background-color: red;  
}
```



13

FONDOS DE IMAGEN

CSS define **6 propiedades** para establecer el fondo de cualquier elemento:

```
background-image: url(..../imagenes/fondo.jpg);  
background-repeat: repeat;  
otros valores posibles → [no-repeat | repeat-x | repeat-y]
```

```
background-position: left top;  
↓      ↓  
Eje X Eje Y
```

otros valores posibles → eje x [left | center | right | length]
otros valores posibles → eje y [top | center | bottom | length]

FONDOS DE IMAGEN

```
background-attachment: scroll;  
otros valores posibles → [fixed | local | initial | inherit]  
  
background-color: #f2f2f2;  
  
background-size: cover;  
otros valores posibles → [contain | length | percentage | inherit]
```

15

14



Módulo: HTML y CSS

FORMULARIOS HTML

1.

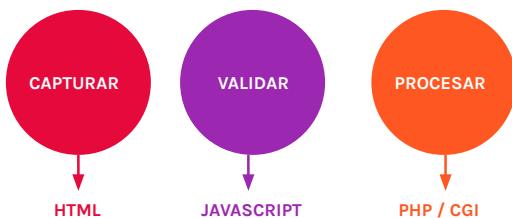
¿QUÉ ES UN FORMULARIO?

Sistema para capturar datos.

Necesita de lenguajes adicionales para su completo funcionamiento.

CÓMO FUNCIONA

Para lograr que un formulario funcione correctamente, hacen falta las siguientes **3 instancias**:



3

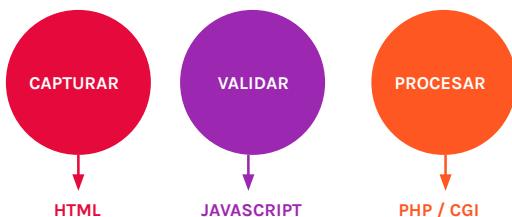
CÓMO FUNCIONA

En la clase de hoy nos preocupamos por la parte de **HTML**, es decir, que el formulario se vea correctamente y que capture la información.

Luego veremos cómo validar y procesar la información

CÓMO FUNCIONA

Para lograr que un formulario funcione correctamente, hacen falta las siguientes **3 instancias**:



5

QUÉ ES LA SEMÁNTICA

Así como la forma de maquetar los sitios webs fue evolucionando con los años, también lo hicieron los **motores de búsqueda**.

El uso correcto de nuestras etiquetas, desde el punto de vista semántico, nos permite reforzar el **significado del contenido de nuestro sitio web**.

De esta forma podemos ser más **específicos** al momento de encerrar contenido entre etiquetas y así crear un código más amigable para los **buscadores**.

SABÍAS QUE...

Incorporar etiquetas semánticas en nuestro código, ayuda mucho al posicionamiento SEO.



6

2.

ELEMENTOS DEL FORMULARIO

ETIQUETA <form>

El tag **más importante**. Sin éste el formulario no funciona. **TODOS** los componentes que queramos incluir van a ir dentro de estas dos etiquetas.

```
<form action="validar.php" method="POST">  
</form>
```

IMPORTANTE

En el aula hablaremos de la importancia de los atributos de esta etiqueta

ETIQUETA <input>

Nos permite generar campos para que el usuario complete con información. Cambiando el valor del atributo type podemos obtener distintos tipos de campos. El atributo name lo identifica y diferencia de los demás campos. name es fundamental para procesar la información del mismo.

```
<input type="text" name="nombre-usuario">  
<input type="email" name="mail">  
<input type="tel" name="telefono">  
<input type="number" name="edad">
```

9

ETIQUETA <label>

Aquella etiqueta que acompaña a un campo. El texto se muestra en el navegador e indica la información que tiene que completar el usuario en el campo.

```
<label>Nombre:</label>  
<label>Email:</label>  
<label>Teléfono:</label>  
<label>Edad:</label>
```

Un formulario más completo

```
<form action="validar.php" method="POST">  
  <p>  
    <label for="nombre">Nombre:</label>  
    <input type="text" name="nombre" id="nombre">  
  </p>  
  <p>  
    <label for="email">Email:</label>  
    <input type="email" name="email" id="email">  
  </p>  
  <p>  
    <input type="submit" value="Enviar">  
  </p>  
</form>
```

11

Así se vería en el navegador

The screenshot shows a web browser window with a light gray header bar containing three colored dots (red, yellow, green) and a search bar. Below the header is a form with two text input fields labeled "Nombre:" and "Email:", each followed by a horizontal input field. Below these fields is a single-line "Enviar" (Send) button.

12



Módulo: HTML y CSS

FORMULARIOS II

1.

ELEMENTO INPUT

Con el atributo **type** podemos generar distintos tipos de **inputs** para nuestro formulario.

¡Veamos de qué se trata!

TYPE RADIO

Seteando el atributo **type** con el valor **radio**, generamos un botón de única opción, más conocidos como **radio-button**.

```
<input type="radio" name="asistio" value="si">
```

Valor para type
Crea el botón.

Valor para name
Para que el usuario pueda seleccionar sólo una opción, tenemos que asignarle el mismo nombre a todos los input de tipo radio que creemos.

Valor para value
Acá debe ir la información que queremos que viaje cuando el usuario seleccione ese botón.

3

TYPE RADIO

Los input de tipo radio se ven a ver de la siguiente manera:

¿Asistió al evento?

Sí
 No

TYPE CHECKBOX

Seteando el atributo **type** con el valor **checkbox**, generamos una casilla de verificación para que el usuario seleccione una o más opciones, más conocidos como **check boxes**.

```
<input type="checkbox" name="hobbies" value="cantar">
```

Valor para type
Crea la casilla de verificación.

Valor para name
Hay que asignarle el mismo nombre a todos los elementos de tipo checkbox que sean del mismo campo.

Valor para value
Acá debe ir la información que queremos que viaje cuando el usuario seleccione esa casilla.

5

TYPE CHECKBOX

Los input de tipo checkbox se ven a ver de la siguiente manera:

Seleccioná tus hobbies:

Fútbol
 Magic
 Andar en bici
 Cantar

6



Si le asignamos el atributo **checked** a un elemento de tipo **input**, ya sea **radio-button** o **checkbox**, éste va a aparecer ya seleccionado.



Módulo: HTML y CSS

FORMULARIOS III

1.

TAGS PARA FORMULARIOS

Conozcamos algunas etiquetas nuevas que nos van a permitir generar más interacción con nuestros usuarios.

CAMPO MULTILÍNEA

Con la etiqueta `<textarea>` creamos un campo para escribir varias líneas de texto. Se suele usar para que el usuario envíe comentarios sobre algo en particular.

```
<textarea name="mensaje"></textarea>
```

3

DESPLEGABLE

Con la etiqueta `<select>` y la etiqueta `<option>` creamos un **combobox** ó **dropdown** de única selección. Esto nos permite agrupar muchas opciones en un solo control. El `<select>` será el contenedor para `<option>`.

```
<select name="pais">
  <option value="Ar">Argentina</option>
  <option value="Br">Brasil</option>
  <option value="Ur">Uruguay</option>
</select>
```

4

BOTONES

Con la etiqueta `<button>` creamos botones. Dependiendo del valor que le asignamos al atributo `type` vamos a poder **enviar**, **borrar** o generar otro tipo de **acción**. Esta etiqueta permite anidamiento de otros tags.

```
<button type="submit">Enviar</button>
<button type="reset">Resetear</button>
<button type="button">Otra acción</button>
```

5



Módulo: HTML y CSS
CSS

1. PSEUDO SELECTORES

Un **pseudo selector** nos permite controlar eventos especiales de un elemento.

Suelen ser aplicados sobre un selector existente.

SINTAXIS

Para escribir nuestro pseudo selector lo podemos hacer de la siguiente manera:

```
Nombre del selector que queremos modificar : Nombre del pseudoselector que va a afectar a mi selector
↑           ↑
selector:pseudoselector {
    propiedad: valor;
}
```

3

:HOVER

Controla el estado “rollover” de cualquier elemento. Lo que definamos con este selector, sólo será visible al posar el cursor sobre ese elemento.

```
a {
    text-decoration: none;
}

a:hover {
    text-decoration: underline;
}
```

Este estilo se aplica cuando el cursor está sobre el elemento

:FOCUS

Controla el estado “focal” de los campos de un formulario. Lo que definamos con este selector, sólo será visible al momento de clickear dentro del campo.

```
input {
    border-color: none;
}

input:focus {
    border-color: green;
}
```

Este estilo se aplica cuando hacemos click sobre el campo

5

:NTH-CHILD()

Nos deja decidir qué etiquetas hermanas queremos afectar, sin importar el elemento padre que tengan. Las seleccionamos con un valor numérico, o también con la palabras **odd** para seleccionar los elementos impares, y **even** para los pares.

```
p {
    color: blue;
}

p:nth-child(2) {
    color: white;
    background-color: orange;
}
```

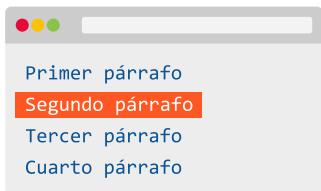
Este estilo se aplica sólo a los elementos que cumplen con la condición entre los paréntesis

6

:NTH-CHILD()

Dado el código HTML de la izquierda, y teniendo en cuenta el selector escrito en la página anterior, el ejemplo se vería de la siguiente manera:

```
<p>Primer párrafo</p>
<p>Segundo párrafo</p>
<p>Tercer párrafo</p>
<p>Cuarto párrafo</p>
```





Módulo: HTML y CSS

ETIQUETAS SEMÁNTICAS

1.

HTML SEMÁNTICO

¿Qué quiere decir?

Maquetar cómo se verá nuestro sitio, pero teniendo en cuenta a los motores de búsqueda.

QUÉ ES LA SEMÁNTICA

El uso correcto de nuestras etiquetas, desde el punto de vista semántico, nos permite reforzar el significado del contenido de nuestro sitio web.

De esta forma podemos ser más específicos al momento de encerrar contenido entre etiquetas y así crear un código más amigable para los buscadores.

SABÍAS QUE...

Incorporar etiquetas semánticas en nuestro código, ayuda mucho al posicionamiento SEO.

3

QUÉ ES LA SEMÁNTICA

Así como la forma de maquetar los sitios webs fue evolucionando con los años, también lo hicieron los motores de búsqueda.

El uso correcto de nuestras etiquetas, desde el punto de vista semántico, nos permite reforzar el significado del contenido de nuestro sitio web.

De esta forma podemos ser más específicos al momento de encerrar contenido entre etiquetas y así crear un código más amigable para los buscadores.

SABÍAS QUE...

Incorporar etiquetas semánticas en nuestro código, ayuda mucho al posicionamiento SEO.



2.

EJEMPLOS DE TAGS SEMÁNTICOS

Conozcamos esas etiquetas que nos van a permitir darle semántica a nuestro sitio.

ETIQUETA ****

A través de esta etiqueta, le estamos diciendo al buscador que el elemento es importante. Por defecto, el texto se verá en negrita.

Texto a destacar

Esta etiqueta convierte al texto en negrita. **NO** es semántico.

Texto a destacar

Esta etiqueta convierte al texto en negrita y le dice al buscador que es importante. **ES** semántica.

ETIQUETA ``

A través de esta etiqueta, le estamos diciendo al buscador que el elemento lleva un **énfasis**. Por defecto el texto se verá en cursiva.

`<i>Palabra importante</i>`

Esta etiqueta convierte al texto en *italic*. **NO** es semántico.

`Palabra importante`

Esta etiqueta convierte al texto en *italic* y le dice al buscador que lleva un énfasis por sobre el resto. **ES** semántica.

ETIQUETA `<mark>`

A través de esta etiqueta, destacamos una parte del texto que sea relevante. Sirve para llamar la atención del lector. Por defecto se va a ver con un fondo de color amarillo.

`<mark>Parte a resaltar</mark>`

7

8

ETIQUETA `<cite>`

Esta etiqueta define el título de una "obra". Ej: un libro, una pintura, una canción, una película, una serie de Tv, etc.

`<cite>Parte del texto a citar</cite>`

ETIQUETA `<abbr>`

Esta etiqueta define la abreviación o acrónimo. Es decir, la versión corta (representada por siglas) de algo con más significado. Al pasar el mouse por encima del elemento, veremos lo que hayamos definido en el atributo title.

`<abbr title="Digital House">DH</abbr>`

9

10



Módulo: HTML y CSS

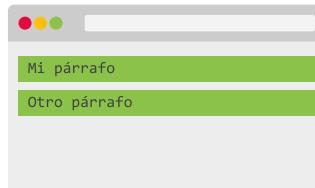
TIPOS DE ELEMENTOS

ETIQUETAS DE BLOQUE

Las etiquetas de **bloque** intenta ocupar el 100% del ancho del sitio. Visualmente generan un salto de línea. Esto se da porque, al ocupar todo el ancho disponible, no dejan espacio para que entre otro elemento. Las etiquetas `<div>` son un ejemplo de etiquetas de bloque muy utilizadas ya que permiten generar **divisiones** en nuestro sitio.

```
<div>Mi párrafo</div>
<div>Otro párrafo</div>

div {
    background-color: green;
}
```



3

1.

EN LÍNEA Y EN BLOQUE

ETIQUETAS EN LÍNEA

Las etiquetas en **línea** ocupan sólo el ancho de su contenido y no cambian la distribución del sitio. Es decir, no van a generar saltos de línea por defecto, ya que su ancho va a estar determinado por el contenido que lleve dentro.

```
<span>Nº 3</span>
<span>Nº 4</span>

span {
    background-color: green;
}
```



4

TIPOS DE ELEMENTOS

inline

Define un elemento con comportamiento en **línea**, y no recibe algunas propiedades del modelo de caja.

block

Define un elemento con comportamiento de **bloque**, y puede recibir propiedades del modelo de caja.

TIPOS DE ELEMENTOS

inline-block

Define un elemento con comportamiento de **semi-bloque**. Puede recibir propiedades del modelo de caja, y también comparte propiedades de elementos de línea.

none

Oculta a un elemento en la visual. No lo elimina de la estructura de HTML, sólo desaparece de la vista.

5

6

TIPOS DE ELEMENTOS

Mediante la propiedad **display** de css podemos cambiar la disposición del elemento que queramos. Los valores que recibe son **block**, **inline**, **inline-block** y **none**.

```
<span>Nº 3</span>
```

```
span {  
    background-color: green;  
    display: block;  
}
```



7

2.

ETIQUETAS SEMÁNTICAS

Lo que verán a continuación
son todas **etiquetas de bloque**

ETIQUETAS SEMÁNTICAS

```
<section></section>
```

Esta etiqueta define una sección de contenido monotemático.

```
<article></article>
```

Esta etiqueta define un fragmento de información dentro de una sección.

ETIQUETAS SEMÁNTICAS

```
<header></header>
```

Esta etiqueta define la cabecera de un contenido determinado o del documento.

```
<footer></footer>
```

Esta etiqueta define el pie de un contenido determinado o del documento.

9

10



Módulo: HTML y CSS

MODELO DE CAJA

1.

¿QUÉ ES?

Seguramente la característica
más importante de CSS

¡DEFINAMOS!

Es el **comportamiento** que hace que todos los elementos de un documento HTML se representen mediante **cajas rectangulares**. De este modo, permite asignarle propiedades a los elementos, y así afectar el alto, el ancho, el margen, etc.

Este modelo condiciona el **diseño de todas** las páginas web.

Las propiedades de modelo de caja solo aplican a **etiquetas de bloque**

PROPIEDAD WIDTH

Si un elemento **no** tiene declarada la **propiedad width**, el ancho será igual al 100% de su padre contenedor, siempre y cuando ése sea un elemento de bloque

Para asignarle **valor** a esta propiedad, lo podemos hacer usando la medida de porcentajes (%) ó píxeles (px).

```
div {  
    width: 120px;  
}
```

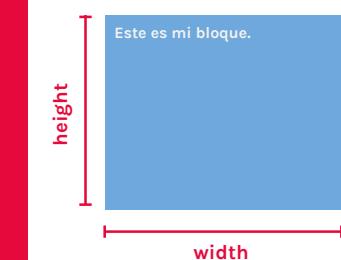
3

PROPIEDAD HEIGHT

Si un elemento **no** tiene declarado la **propiedad height**, el alto será igual a la altura que le proporcione su contenido interno. Sea un elemento de bloque o de línea.

Para asignarle **valor** a esta propiedad, lo podemos hacer usando la medida píxeles (px).

```
div {  
    height: 130px;  
}
```



```
div {  
    background-color: blue;  
    width: 120px;  
    height: 130px;  
}
```

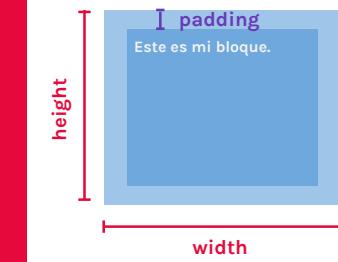
5

6

PROPIEDAD PADDING

Hace referencia al **margin interior** del elemento. Para asignarle **valor** a esta propiedad, lo podemos hacer usando la medida píxeles (px), indicando **1 valor** para los 4 lados de la caja. También podemos hacerlo con **2 valores**, el primero va a indicar el padding de arriba y abajo, y el segundo el de la izquierda y la derecha.

```
div {  
    padding: 12px;  
}  
  
div {  
    padding: 22px 30px;  
}
```



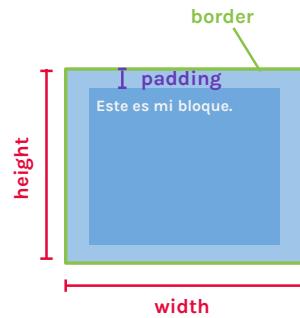
```
div {  
    background-color: blue;  
    width: 120px;  
    height: 130px;  
    padding: 12px;  
}
```

PROPIEDAD BORDER

Hace referencia al borde del elemento. Para asignarle **valor** a esta propiedad, lo hacemos definiendo el **estilo de línea**, su **tamaño** y su **color**. El estilo de línea puede ser solid, dotted, dashed o double.

```
div {  
    border: solid 3px green;  
}
```

Estilo de línea Color
↑ ↑
↓
Tamaño

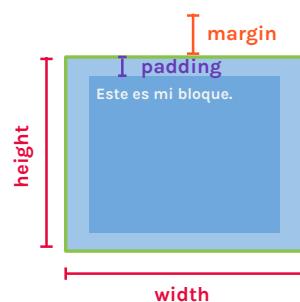


```
div {  
    background-color: blue;  
    width: 120px;  
    height: 130px;  
    padding: 12px;  
    border: solid 3px green;  
}
```

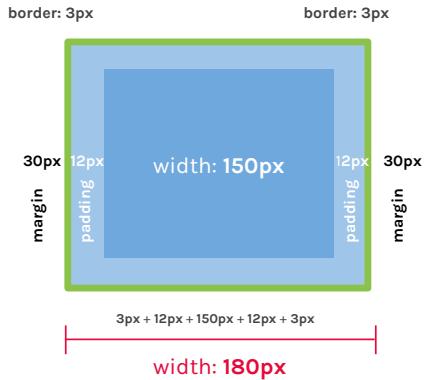
PROPIEDAD MARGIN

Hace referencia al **margin** del elemento. Sirve para separar una caja de la otra. Para asignarle **valor** a esta propiedad, lo podemos hacer usando la medida píxeles (px), indicando **1 valor** para los 4 lados de la caja. También podemos hacerlo con **2 valores**, el primero va a indicar el padding de arriba y abajo, y el segundo el de la izquierda y la derecha.

```
div {  
    margin: 30px;  
}
```



```
div {  
    background-color: blue;  
    width: 150px;  
    height: 130px;  
    padding: 12px;  
    border: dotted 3px green;  
    margin: 30px;  
}
```



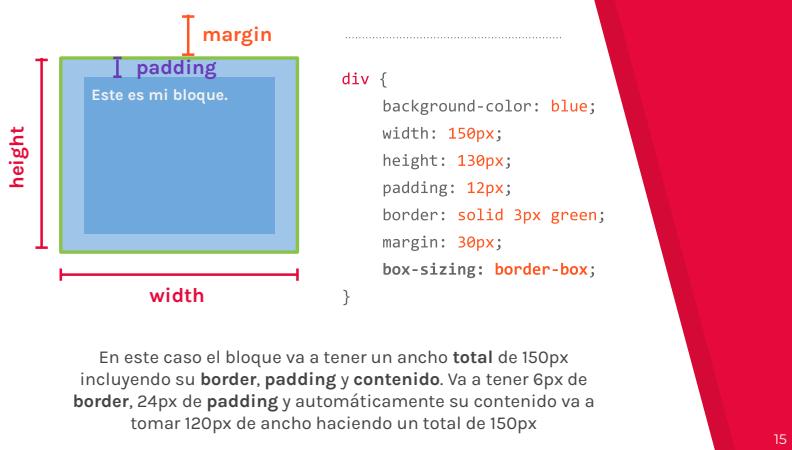
El ancho total de la caja es igual a la suma de su **width**, **padding** y **border**. Además cuenta con 60px de **margin**

PROPIEDAD BOX-SIZING

Esta propiedad permite que el modelo de caja sea más fácil de usar, porque descuenta automáticamente del ancho y alto lo que agregamos en relleno y borde. El único valor que se sigue sumando es el del **margin**.

13

14



15

16

“

INTRODUCCIÓN A FLEXBOX



Es una metodología de **CSS** que permite **maquetar** un sitio web utilizando una **estructura** de **filas y columnas**.

INICIOS DE FLEXBOX

CSS (Cascade Style Sheet) nace en el año **1994**, y desde ese entonces fueron pocas las nuevas implementaciones que recibió el lenguaje.

Recién a mediados del **2008** se empieza a discutir la posibilidad de implementar una **nueva forma de maquetación**, que implique una **estructura** de cajas flexibles.

En el año **2011** salió a la luz el primer borrador con las especificaciones para implementar **Flexbox**. Al ser una **técnica nueva**, los navegadores en su versión comercial todavía no le aportaban suficiente **soporte**.

La W3C **aceptó** y **oficializó** la implementación de Flexbox en el lenguaje.

Los **desarrolladores** recibieron esto como un **cambio** muy **positivo**, entendiendo que les proporcionaba más libertad que el posicionamiento por **flotación**.



3

VENTAJAS DE FLEXBOX

Cuando usamos **flotación** para **posicionar** un elemento en un sitio web, el mismo deja de formar parte del **flujo** natural de la **estructura de elementos**. Esto genera solapamiento de cajas y estructuras difíciles de mantener.

Flexbox propone un **único flujo**, en el que dispondremos de los elementos con mayor libertad para **distribuir**, **redimensionar** y **reordenar** cada uno de ellos en función de ese flujo de trabajo.



En el **2015** fue oficialmente adoptado por las versiones comerciales de todos los **navegadores web**.

Actualmente tiene un **soporte** de más del **98%** en las **distintas versiones** de cada uno de ellos.



5

“

ESTRUCTURA BÁSICA DE FLEXBOX

Esta metodología propone una estructura basada en el uso de un **contenedor padre** (*Flex-container*) y sus **elementos hijos** (*Flex-items*).



TRABAJAR CON FLEXBOX

Para empezar a trabajar con flexbox tenemos que definir un **flex-container**. Para eso usamos la propiedad `display` con el valor `flex`. De esta forma estamos habilitando un **contexto flex** para trabajar con los **hijos directos** del elemento.

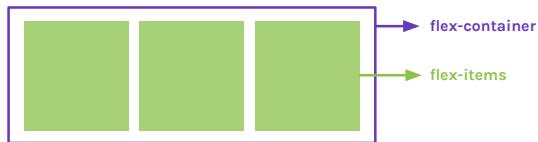
```
css
.contenedor-padre {
  display: flex;
}
```

La propiedad `display` también puede recibir el valor `inline-flex`

ESTRUCTURA BÁSICA

Cuando hablamos de un **flex-container**, hablamos de un elemento HTML que **contiene** a uno o más elementos. A estos elementos anidados, los llamamos **flex-items**.

Es en el **flex-container** en donde configuramos la mayoría de las propiedades flex.



flex-wrap

Por defecto, los elementos hijos de un contenedor flex, van a tratar de **entrar todos en una misma línea**. Para aclararle al contenedor que **debe respetar** el **ancho** definido de sus **hijos**, usamos la propiedad `flex-wrap` con el valor `wrap`.

```
css
.contenedor-padre {
  display: flex;
  flex-wrap: wrap;
}
```

La propiedad `flex-wrap` también puede recibir los valores `nowrap` y `wrap-reverse`.

Un **flex-item**, a su vez, puede convertirse en un **flex-container**.

Para eso, sólo hace falta asignarle la regla `display:flex`, para que así sus **elementos hijos** pasen a ser **flex-items**.



“

Flexbox trabaja con dos ejes para **desarrollar** todo su **flujo** interno: el **eje X** y el **eje Y**.



LOS EJES

EJES EN FLEXBOX

Dentro de un flex-container, tanto el **eje x** como el **eje y** toman otros nombres. Cuando trabajamos en un flujo flex hablamos del **main axis** y el **cross axis**.

El concepto de trabajo de flexbox está basado en una sola dirección, es decir, los elementos se distribuyen o en **filas horizontales** o en **columnas verticales**.

Definiendo el **eje principal** de nuestro contenedor flex, estamos determinando el flujo que tendrán los elementos dentro del contenedor. Los mismos se ordenarán en base al eje que definamos.

flex-direction

Con esta propiedad **definimos** el **main axis** del contenedor (**eje principal**), que puede ser tanto **horizontal** como **vertical**. El **cross axis** (**eje transversal**), será la dirección perpendicular al main axis.



3

flex-direction: row

`flex-direction: row`

Los ítems se disponen en el **eje x**, de **izquierda a derecha**.

Si no le aclaramos la propiedad `flex-direction` al contenedor, `row` es el valor por defecto.



flex-direction: row-reverse

`flex-direction: row-reverse`

Los ítems se disponen en el **eje x**, de **derecha a izquierda**.



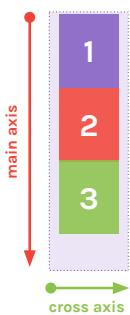
5

6

flex-direction: column

flex-direction: column

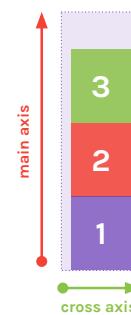
Los ítems se disponen en el **eje y**, de **arriba hacia abajo**.



flex-direction: column-reverse

flex-direction: column-reverse

Los ítems se disponen en el **eje y**, de **abajo hacia arriba**.



7

8

Flexbox nos da **dos propiedades** para **alinear** fácilmente los elementos tanto a través del **main axis** como del **cross axis**.



justify-content

Con esta propiedad **alineamos** los **ítems** a lo largo del **main axis**. Si es horizontal, se alinearán en función de la fila. Si es vertical, se alinearán en función de la columna.

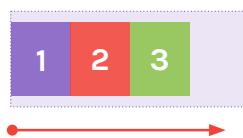


justify-content: flex-start

justify-content: flex-start

Los ítems se alinean respecto del **inicio** del **main axis** que hayamos definido.

Si no le aclaramos el justify-content al contenedor, flex-start es el valor por defecto.



justify-content: flex-end

justify-content: flex-end

Los ítems se alinean respecto del **final** del **main axis** que hayamos definido.



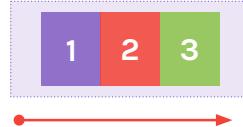
11

12

justify-content: center

justify-content: center

Los ítems se alinean en el **centro** del main axis.



justify-content: space-between

justify-content: space-between

Los ítems se **distribuyen de manera uniforme**. El primer ítem será enviado al **inicio** del main axis, el último ítem será enviado al **final** del main axis.



13

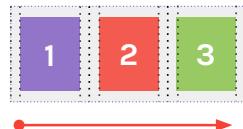
14

justify-content: space-around

justify-content: space-around

Los ítems se **distribuyen de manera uniforme**, con igual espacio alrededor de cada uno.

El primer ítem tendrá **una unidad** de espacio contra el borde del contenedor, y **dos unidades** de espacio contra el siguiente ítem, porque el mismo tiene su **propio espacio** que se aplica. Lo mismo sucede con el **último ítem**.



15

align-items

Con esta propiedad **alineamos** los **ítems** a lo largo del **cross axis**. Si no aclaramos esta propiedad, el valor por defecto es **stretch**.



align-items: stretch

align-items: stretch

Los ítems se **ajustan** para abarcar todo el contenedor. Si el **cross axis** es **vertical**, se ajustan en función de la **columna**. Si el **cross axis** es **horizontal**, se ajustan en función de la **fila**.



align-items: flex-start

align-items: flex-start

Los ítems se alinean al **inicio** del eje transversal.



17

18

align-items: flex-end

align-items: flex-end

Los ítems se alinean al final del eje transversal.



align-items: center

align-items: center

Los ítems se alinean al centro del eje transversal..



“

Flexbox nos da la posibilidad de aplicarle **propiedades** directamente a cada **ítem** para poder **manipularlos** por **separado** y tener más control.



LOS ÍTEMES

order

Con esta propiedad **controlamos** el **orden** de cada ítem, sin importar el orden original que tengan en la estructura HTML.



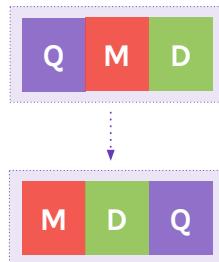
Esta propiedad recibe un **número entero** como **valor**.

order: número positivo

Si le asignamos a la caja Q la propiedad `order` con valor 1, la misma pasará al final de la fila por ser el número más alto.

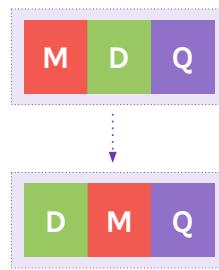
Por defecto, el valor del orden de cada ítem es 0.

```
.cajaQ {  
    order: 1;  
}
```



order: número negativo

Si ahora le asignamos a la caja D la propiedad `order` con un -1 como valor, la misma pasará al principio de la fila.



```
.cajaD {  
    order: -1;  
}
```

Las cajas se irán **ordenando** **respetando** la secuencia desde los números **negativos** hacia los **positivos**.



flex-grow

Con esta propiedad definimos cuánto puede llegar a **crecer** un ítem en caso de **disponer** de **espacio libre** en el contenedor.

Configura un crecimiento flexible para el elemento.



flex-grow

Si **ambos ítems** tienen la propiedad `flex-grow` con valor 1, a medida que el contenedor se agrande, irán abarcando el espacio disponible en partes iguales.



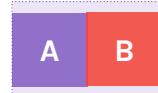
```
css  
.cajaA, .cajaB {  
    flex-grow: 1;  
}
```



flex-grow

Si un solo ítem tiene la propiedad `flex-grow`, el mismo intentará ocupar el espacio libre disponible a medida que el contenedor se agrande, según la proporción que definamos con el valor.

```
css  
.cajaB {  
    flex-grow: 1;  
}
```



El número que le asignamos a **flex-grow** determina qué cantidad de espacio disponible dentro del contenedor flexible tiene que ocupar ese ítem.



align-self

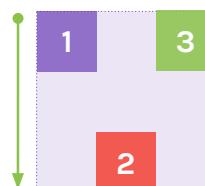
Nos permite **alinear**, sobre el **cross axis**, a cada ítem al que le **apliquemos** esta propiedad, independientemente de la **alineación** que se haya definido en el **contenedor flex** con **align-items**.



align-self: flex-end

Con `flex-end` el ítem se alinea al **final** del eje transversal.

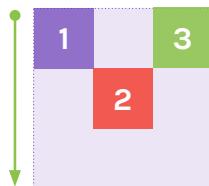
```
css  
.contenedorPadre {  
    align-items: flex-start;  
}  
.caja2 {  
    align-self: flex-end;  
}
```



align-self: center

Con `center` el ítem se alinea al **centro** del eje transversal.

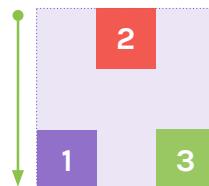
```
css  
.contenedorPadre {  
  align-items: flex-start;  
}  
.caja2 {  
  align-self: flex-end;  
}
```



align-self: flex-start

Con `flex-start` el ítem se alinea al **inicio** del eje transversal.

```
css  
.contenedorPadre {  
  align-items: flex-end;  
}  
.caja2 {  
  align-self: flex-start;  
}
```



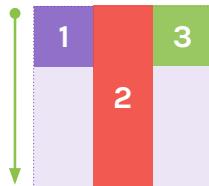
13

14

align-self: stretch

Con `stretch`, el ítem se **ajusta** hasta abarcar todo el **cross axis**, siempre y cuando no tenga definida una altura y el contenedor padre tenga la regla `flex-wrap: wrap`.

```
css  
.caja2 {  
  align-self: stretch;  
}
```



Estas propiedades aplicarán para los flex-items **siempre y cuando** el **padre contenedor** sea un flex-container.



15



Módulo: HTML y CSS

POSICIONAMIENTO

TRASLADAR LOS ELEMENTOS

El posicionamiento nos permite trasladar un elemento desde su posición original a una nueva posición.

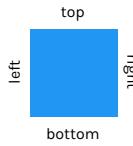
También nos permite superponer elementos.



PUNTOS DE REFERENCIA

Cada uno de los elementos de nuestra página web tiene cuatro puntos de referencia y esos son sus costados.

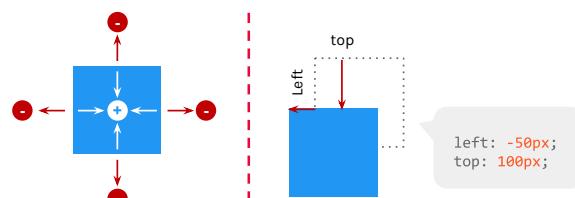
- ↑ Arriba (top)
- Derecha (right)
- ↓ Abajo (bottom)
- ← Izquierda (left)



3

PUNTOS DE REFERENCIA

Cuando desplazamos un elemento tomando un costado como referencia, el movimiento será **positivo si empujamos** el elemento y **negativo si tiramos** de él.



1.

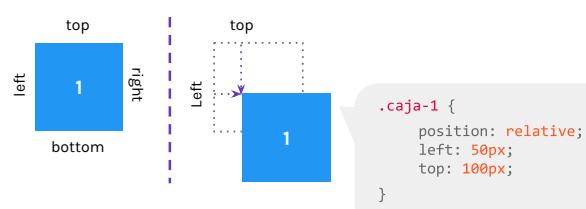
POSICIONAMIENTO RELATIVO

El **posicionamiento relativo** nos permite **trasladar un elemento** desde su posición original a una nueva posición.

Siempre **tomando como referencia** sus propios costados.

POSICIONAMIENTO RELATIVO

Como dijimos, cuando movemos una caja, el punto de referencia serán sus propios costados.



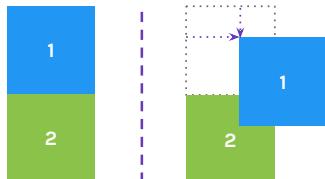
2

4

6

POSICIONAMIENTO RELATIVO

Cuando movemos una caja de manera relativa, el espacio que ocupaba originalmente la caja seguirá ocupado.

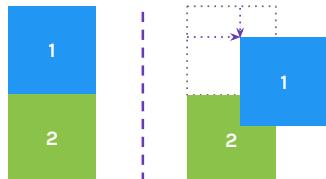


```
.caja-1 {  
    position: relative;  
    left: 100px;  
    top: 50px;  
}
```

7

¿CUÁNDΟ LO USAMOS?

Cuando queremos desplazar un elemento, sin modificar el flujo original.



```
.caja-1 {  
    position: relative;  
    left: 100px;  
    top: 50px;  
}
```

8

2.

POSICIONAMIENTO ABSOLUTO

El posicionamiento absoluto nos permite trasladar un elemento desde su posición original a una nueva posición.

En este caso tomando como referencia los costados del body.

POSICIONAMIENTO ABSOLUTO

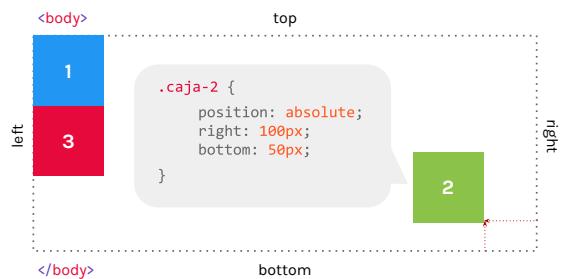
Cuando movemos una caja de manera absoluta, el espacio que ocupaba quedará vacío y otros elementos podrán ocuparlo.



11

POSICIONAMIENTO ABSOLUTO

Con el posicionamiento absoluto, los puntos de referencia serán los costados del body.



</body>

top

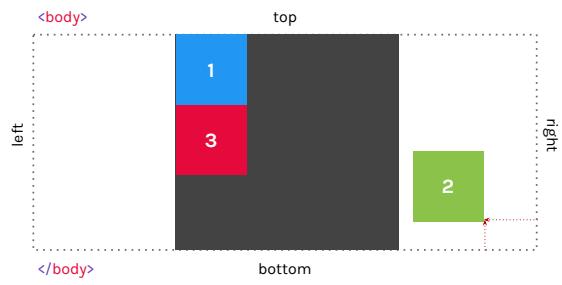
left

right

bottom

POSICIONAMIENTO ABSOLUTO

Aún si nuestras cajas están dentro de otra caja, el punto de referencia seguirá siendo el body.



</body>

top

left

right

bottom

9

10

12

RELATIVO + ABSOLUTO

Si queremos cambiar el punto de referencia para un posicionamiento absoluto, debemos hacer relativo a su padre.

The diagram illustrates the relationship between relative and absolute positioning. A dark grey container labeled ".contenedor" has "position: relative;" applied. Inside, a red box labeled "1" is positioned at the top left. To its right is a green box labeled "2". Below box 1 is a blue box labeled "3". The container has "top", "bottom", "left", and "right" boundaries. A separate code block shows ".caja-2 { position: absolute; right: 100px; bottom: 50px; }". Another code block shows ".contenedor { position: relative; }". The page number 13 is at the bottom right.

¿CUÁNDΟ LO USAMOS?

Cuando queremos sacar un elemento del flujo normal y posicionarlo en un punto fijo con respecto a su contenedor o el body.

The diagram shows a browser window with "HTML" and "CSS" logos. A video player icon is positioned absolutely within the window. A code block shows ".video { position: relative; }" and ".duration { position: absolute; right: 15px; bottom: 15px; }". The page number 14 is at the bottom right.

3. POSICIONAMIENTO FIJO

El posicionamiento fijo nos permite trasladar un elemento desde su posición original a una nueva posición.

En este caso tomando como referencia la ventana del navegador.

POSICIONAMIENTO FIJO

Con el posicionamiento fijo, los puntos de referencia serán los costados la ventana del navegador.

The diagram shows a browser window with four colored boxes (1, 2, 3, 4) and a green box "2" positioned absolutely. A code block shows ".caja-2 { position: fixed; right: 25px; bottom: 0px; }". The page number 16 is at the bottom right.

POSICIONAMIENTO FIJO

Sin importar que hagamos scroll en la página el elemento siempre se mantendrá fijo con respecto a la ventana del navegador.

The diagram shows a browser window with four colored boxes (1, 2, 3, 4) and a green box "2" positioned absolutely. A code block shows ".caja-2 { position: fixed; right: 25px; bottom: 0px; }". The page number 17 is at the bottom right.

¿CUÁNDΟ LO USAMOS?

Cuando queremos que un elemento siga al usuario a medida que navega nuestro sitio. Por ejemplo una ventana de chat.

The diagram shows a browser window with a user profile icon and a message input field. A code block shows ".chat { position: fixed; right: 25px; bottom: 0px; }". The page number 18 is at the bottom right.

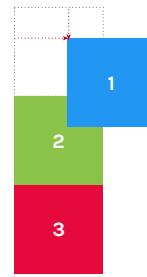
1.

Z-INDEX

El Z-INDEX permite cambiar el orden de las “capas” dentro de un documento HTML.

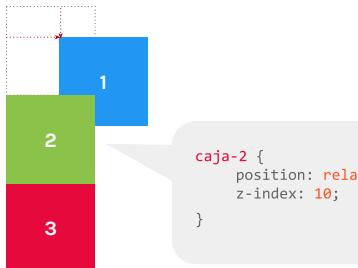
Sólo funciona si el elemento tiene asignado posicionamiento relativo, absoluto o fijo.

EJEMPLO DE Z-INDEX



```
caja-1 {  
    position: relative;  
    left: 100px;  
    top: 50px;  
}
```

EJEMPLO DE Z-INDEX



Siempre va a figurar adelante el elemento que tenga mayor z-index. Por defecto todos tienen z-index en 0.

```
caja-2 {  
    position: relative;  
    z-index: 10;  
}
```



Módulo: HTML y CSS
VIEWPORTS

1. ETIQUETA VIEWPORT

La etiqueta `<meta name="viewport"` da al browser instrucciones de cómo se debe dimensionar y escalar la página web al cargarse.

CONFIGURANDO LA ETIQUETA

Esta es la estructura básica de este tag. Asimismo, en ocasiones puede que lo veamos con más información.

```
<meta name="viewport"  
content="width=device-width, initial-scale=1">
```

3

CONFIGURANDO LA ETIQUETA



SIN etiqueta viewport



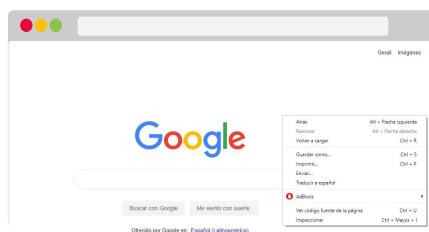
CON etiqueta viewport

2. DEV TOOLS

Nos permiten inspeccionar el código de cualquier sitio y realizar modificaciones sobre el mismo.

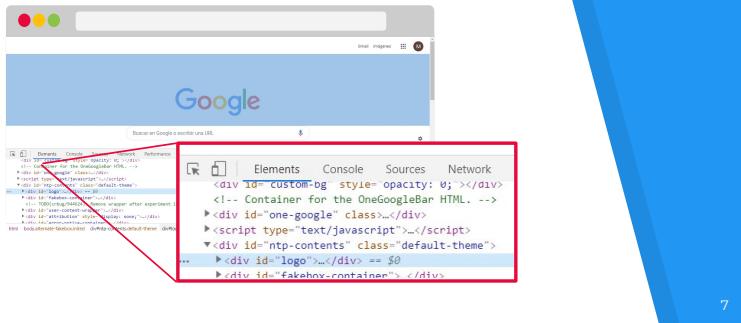
IMPORTANT! Esos cambios, los vemos nosotros. Una vez que refrescamos el sitio, vuelve todo a verse como estaba.

CÓMO ACCEDEMOS



Con click derecho sobre el navegador y luego inspeccionar.
Atajo → F12

CÓMO ACCEDEMOS



7



Módulo: HTML y CSS

MEDIDAS RELATIVAS

1. ¿QUÉ SON?

Son aquellas medidas que están relacionadas con su parente contenedor directo.

PORCENTAJES - %

Cualquier medida expresada en porcentaje, **SIEMPRE** estará relacionada con la medida (en ese mismo eje) del elemento padre que la contiene.

Los altos no se trabajan en medidas porcentuales.

```
.elementoPadre {  
    width: 300px;  
}  
  
.elementoHijo {  
    width: 50%;
```

En este caso, el elemento medirá 150px, siendo el 50% del ancho de su contenedor padre.

3

¿Y CÓMO
SABEMOS QUÉ
PORCENTAJE
ASIGNARLE A
CADA ELEMENTO?



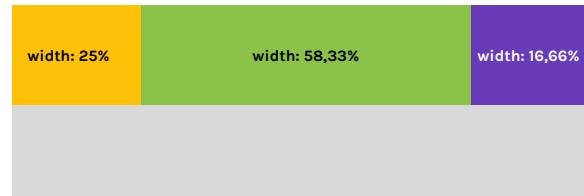
REGLA DE 3 SIMPLE



Contenedor padre width: 1200px

5

REGLA DE 3 SIMPLE



Contenedor padre max-width: 1200px

6

EM'S - em

Los em's son unidades de medida que se recomiendan usar en todo lo relacionado con tipografía.

1em comenzará siendo igual a 16px (a menos que configuremos lo contrario)



Toda medida (que no sea el font-size) expresada en em's tomará SIEMPRE como base referencial el font-size que tenga el mismo elemento que estemos modificando.

7

EM'S - em

<body> tiene 16px en font-size, los demás elementos por default vienen con 1em en font-size. Por lo tanto, todos los demás elementos tienen 16px de font-size.

```
<body> —— Font-size: 16px;  
  <div> —— Font-size: 1em;  
    <p> —— Font-size: 1em;  
      <strong>Hola!</strong> —— Font-size: 1em;  
    <p>  
  <div>  
</body>
```

8

EM'S - em

Si cambiamos el font-size de <div> a 2em, éste tendrá 2 veces el tamaño del font-size de su padre. <div> ahora tiene 32px de font-size. Por lo tanto los hijos de <div> ahora cambiaron con solo haber tocado a su padre. Los hijos de <div> quedaron con 32px en font-size.

```
<body> —— Font-size: 16px;  
  <div> —— Font-size: 2em;  
    <p> —— Font-size: 1em;  
      <strong>Hola!</strong> —— Font-size: 1em;  
    <p>  
  <div>  
</body>
```

9

EM'S - em

<div> tiene 32px de font-size. Si a <p> le decimos font-size: 1.5em, <p> tendrá 1.5 veces el tamaño que tenga el font-size de su padre. <p> ahora tiene 48px en font-size. tiene 1 vez lo que tiene su padre en font-size. tiene 48px en font-size.

```
<body> —— Font-size: 16px;  
  <div> —— Font-size: 2em;  
    <p> —— Font-size: 1.5em;  
      <strong>Hola!</strong> —— Font-size: 1em;  
    <p>  
  <div>  
</body>
```

10

EM'S - em

Si ahora cambiamos la base del documento (<body>) y la llevamos a 10px en font-size, todos los elementos cambian de tamaño sin haber "tocado" su font-size. <div> = 20px de font-size, <p> = 30px de font-size, = 30px de font-size.

```
<body> —— Font-size: 10px;  
  <div> —— Font-size: 2em;  
    <p> —— Font-size: 1.5em;  
      <strong>Hola!</strong> —— Font-size: 1em;  
    <p>  
  <div>  
</body>
```

11

2. VIEWPORT MEASURES

Viewport hace referencia a la "caja visible" de contenido dentro de un navegador.

VIEWPORT



VIEWPORT



vw / vh

Cualquier medida expresada en **viewport width (vw)** ó **viewport height (vh)** tomará SIEMPRE como eje referencial al viewport del documento.



15



Módulo: HTML y CSS

MEDIA QUERIES

1. ¿QUÉ SON?

Son un conjunto de reglas CSS que permiten **reorganizar** el contenido dependiendo de las condiciones de visualización del documento.

Siempre se deben escribir al final de nuestra hoja de CSS.

MIN-WIDTH

```
@media (min-width: 460px){  
  body {  
    background: red;  
  }  
}
```

Al especificar **min-width**, estamos diciendo "si como mínimo hay Npx de ancho, apliquemos estas reglas".
Similar a decir → *Desde este ancho, hacia arriba.*

3

MAX-WIDTH

```
@media (max-width: 460px){  
  body {  
    background: red;  
  }  
}
```

Al especificar **max-width**, estamos diciendo "si como máximo hay Npx de ancho, apliquemos estas reglas".
Similar a decir → *Desde este ancho, hacia abajo.*

4

ORIENTACIÓN

```
@media (max-width: 460px) and (orientation: landscape){  
  body {  
    background: red;  
  }  
}
```

Al especificar la orientación (portrait o landscape), estamos diciendo "si como máximo hay Npx de ancho y además el dispositivo está en posición vertical/horizontal, apliquemos estas reglas".

5

2. ESTRATEGIAS DE DISEÑO

MOBILE FIRST

```
body {  
background: red;  
...  
}  
  
@media (min-width: 460px){  
//tablets  
}  
  
@media (min-width: 460px){  
//desktop  
}
```

1º → Defino los estilos para **mobile**

2º → Defino los estilos para que se adapte en cada viewport

7

MOBILE LAST

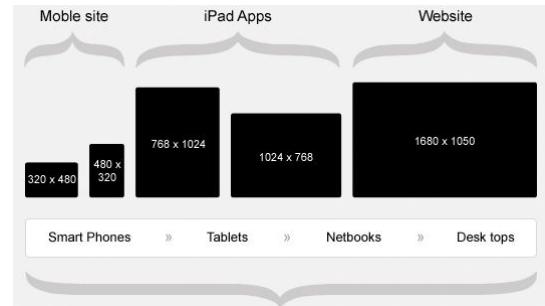
```
body {  
background: red;  
...  
}  
  
@media (max-width: 460px){  
//tablets  
}  
  
@media (max-width: 460px){  
//mobiles  
}
```

1º → Defino los estilos para **escritorio**

2º → Defino los estilos para que se adapte en cada viewport

8

3. ESTÁNDARES



10

“

RUTAS EN EXPRESS

A través del **sistema de ruteo** de Express podemos definir, de manera sencilla, cómo va a responder nuestra aplicación según el **método HTTP** y la **ruta** que esté llegando al **servidor**.



DEFINIENDO UNA RUTA

A través de Express, contamos con una **estructura básica** para definir cada una de las rutas y sus respuestas de nuestra aplicación:

```
app.get('/', function (req , res) {
  res.send('¡Hola mundo!');
})
```

3

DEFINIENDO UNA RUTA

A través de Express, contamos con una **estructura básica** para definir cada una de las rutas y sus respuestas de nuestra aplicación:

```
app.get('/', function (req , res) {
  res.send('¡Hola mundo!');
})
```

Método

Escribimos el **método HTTP** que queremos atender:
get, post, put, patch ó delete.

5

DEFINIENDO UNA RUTA

A través de Express, contamos con una **estructura básica** para definir cada una de las rutas y sus respuestas de nuestra aplicación:

```
app.get('/', function (req , res) {
  res.send('¡Hola mundo!');
})
```

Variable que guarda la **ejecución** de Express.

4

DEFINIENDO UNA RUTA

A través de Express, contamos con una **estructura básica** para definir cada una de las rutas y sus respuestas de nuestra aplicación:

```
app.get('/' function (req , res) {
  res.send('¡Hola mundo!');
})
```

Path

String que hará referencia a la **ruta** en sí (url que llegará por petición).

6

DEFINIENDO UNA RUTA

A través de Express, contamos con una **estructura básica** para definir cada una de las rutas y sus respuestas de nuestra aplicación:

```
app.get('/', function (req , res) {  
    res.send('¡Hola mundo!');  
})
```

Handler

Callback que se encargará de definir qué acción tomar cuando se acceda a la ruta definida.

DEFINIENDO UNA RUTA

A través de Express, contamos con una **estructura básica** para definir cada una de las rutas y sus respuestas de nuestra aplicación:

```
app.get('/', function (req , res) {  
    res.send('¡Hola mundo!');  
})
```

Request (primer parámetro del handler)

Es un objeto literal con múltiples métodos y propiedades. Representa al **request** solicitado.

7

DEFINIENDO UNA RUTA

A través de Express, contamos con una **estructura básica** para definir cada una de las rutas y sus respuestas de nuestra aplicación:

```
app.get('/', function (req , res) {  
    res.send('¡Hola mundo!');  
})
```

Response (segundo parámetro del handler)

Es un objeto literal con múltiples métodos y propiedades. Representa al **response** que dará el servidor.

DEFINIENDO UNA RUTA

A través de Express, contamos con una **estructura básica** para definir cada una de las rutas y sus respuestas de nuestra aplicación:

```
app.get('/', function (req , res) {  
    res.send('¡Hola mundo!');  
})
```

Lógica de la ruta

Definimos la **lógica** que va a manejar la ruta definida. Se estila dar la respuesta que verá el cliente en su navegador.

9

8

10

RUTAS PARAMETRIZADAS

Express nos permite crear **rutas dinámicas** en las que definimos qué parámetro es el que va a ir variando.



RUTAS PARAMETRIZADAS

Usando la misma **estructura básica** para definir una ruta, aclaramos en el **path**, cuál es el parámetro que va a ir variando haciendo uso de los dos puntos `:`, seguido del **nombre** que represente al dato que estará llegando en la url.

```
app.get('/productos/:id', function (req,res) {
  // código
})
```

RUTAS PARAMETRIZADAS

Usando la misma **estructura básica** para definir una ruta, aclaramos en el **path** cuál es el parámetro que va a ir variando haciendo uso de los dos puntos `:`, seguido del **nombre** que represente al dato que estará llegando en la url.

```
app.get('/productos/:id', function (req,res) {
  // código
})
```

Parámetro obligatorio

Definimos el parámetro variable que sí o sí va a llegar a través de la url.

RUTAS PARAMETRIZADAS

Usando la misma **estructura básica** para definir una ruta, aclaramos en el **path** cuál es el parámetro que va a ir variando haciendo uso de los dos puntos `:`, seguido del **nombre** que represente al dato que estará llegando en la url.

```
app.get('/productos/:id?', function (req,res) {
  // código
})
```

Parámetro optativo

Definimos el parámetro variable que puede o no llegar a través de la url, agregando un signo de interrogación al final.

RUTAS PARAMETRIZADAS

Haciendo uso de la propiedad `params` del objeto literal `request`, podemos **capturar** esos valores parametrizados y así empezar a definir qué hacer con cada uno de ellos.

Esta propiedad, a su vez, es un **objeto literal** que guarda los parámetros que llegan por url con la estructura `propiedad:valor`, en donde cada **propiedad** tendrá el **nombre** del parámetro que definimos en el **path**.

```
app.get('/productos/:id', function (req,res) {
  let idProducto = req.params.id;
})
```

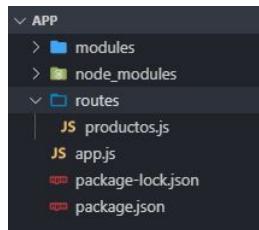
SISTEMA DE RUTEO

Es importante plantear y mantener un **orden estructural** cuando desarrollamos.



ARQUITECTURA DE ARCHIVOS

En la carpeta **routes** guardaremos, por cada recurso, un archivo js que administre los **request** a las rutas que tengan que ver con ese recurso.
Cada archivo js será un **módulo** que exportaremos, para luego requerir dentro del entry point de la aplicación: `app.js`



3

SISTEMA DE RUTEO

Para empezar a definir el sistema de ruteo de cada recurso, hace falta situarse dentro del **archivo js** del recurso, requerir el **módulo express** y guardar la ejecución del método `Router()` que nos provee Express en una variable nueva.

```
const express = require('express');
const router = express.Router();
```

Este método nos va a permitir **modularizar** por completo todo el **sistema de ruteo** de una manera sencilla.

SISTEMA DE RUTEO

Definimos las rutas que consideremos necesarias para manejar distintos tipos de **request**.

```
// Ruta raíz de los productos / Inicio
router.get('/', (req, res) => {
  // código
});

// Ruta que muestra el detalle de un producto
router.get('/detalle/:id', (req, res) => {
  // código
});
```

5

SISTEMA DE RUTEO

En la última línea del archivo **exportamos** todo el contenido de **router** para hacerlo visible.

```
module.exports = router;
```

Para implementarlo dentro de `app.js`, creamos una constante y requerimos el módulo.

```
const rutasProductos = require('./routes/productos');
```

4

6

SISTEMA DE RUTEO

Por último, hacemos uso del método `use()` que recibe dos parámetros. El primero un string que será el **nombre** del **recurso**, en este caso `productos`. Al ser una ruta debe empezar con la `/`.

El segundo el nombre de la constante en la que almacenamos el módulo del recurso.

```
{ } app.use('/productos', rutasProductos);
```

De esta forma estamos definiendo que cada solicitud del recurso **productos**, sea atendida por el módulo **rutasProductos** y toda su lógica.

INTRODUCCIÓN A MVC

Es un **patrón de diseño**.
Sus siglas corresponden a
Modelo Vista Controlador.



QUE ES UN PATRÓN DE DISEÑO

Dentro del mundo de la programación existen lo que se conocen como Patrones de Diseño. Los mismos proponen un **esquema de trabajo**, una serie de **reglas** que permiten simplificar el código y encarar mejor la solución de diferentes situaciones a lo largo del desarrollo.

Uno de los patrones más populares es **MVC**. Su objetivo es crear aplicaciones modulares, dividiendo la columna vertebral del proyecto en **tres componentes** principales, en donde cada uno de ellos cumple con un rol determinado. Estos componentes son: los **modelos**, las **vistas** y los **controladores**.

LAS VISTAS

Conforman la **interfaz gráfica** de la aplicación y contienen todos los elementos que son visibles al usuario. A través de ellas el usuario interactúa enviando y solicitando información al servidor.

Su responsabilidad es **definir la apariencia** de los datos y **mostrarlos** en pantalla.

Las **vistas** **no** se comunican de forma directa con los **modelos**.



3

4

LOS MODELOS

Conforman y contienen la **lógica** de la aplicación.

Sus responsabilidades son **conectarse** con la base de datos, realizar **consultas** y **administrar** lo que se conoce como la lógica de negocio.

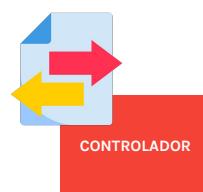
Los **modelos** **no** se comunican de forma directa con las **vistas**.



LOS CONTROLADORES

Conforman la capa intermedia entre las **vistas** y los **modelos**.

Su responsabilidad es **procesar** los datos que recibe de los **modelos** y **elección** la vista correspondiente en función de aquellos datos.



Tienen **relación directa** con las **vistas** y con los **modelos** y es un componente fundamental dentro del flujo del patrón.

5

6

EJEMPLO FLUJO MVC



Un usuario recorre, a través de la interfaz gráfica, un listado de productos y quiere solicitar más información acerca del producto 20. La **vista** entonces se conecta con el **controlador** para solicitarle esos datos.

EJEMPLO FLUJO MVC



El **controlador** recibe la petición y le solicita al **modelo** el detalle del producto 20.

7

8

EJEMPLO FLUJO MVC



El **modelo** busca la información solicitada y se la envía al **controlador**.

EJEMPLO FLUJO MVC



El **controlador** recibe la información y le envía los datos a la **vista**.

9

10

EJEMPLO FLUJO MVC



La **vista** le muestra al usuario los datos que recibió.

Es importante destacar que los patrones de diseño **proponen** una forma de trabajo y no hay nadie por detrás que regule que se cumplan tales formas.

Se utilizan como una **buenas prácticas** que facilita el trabajo en equipo, permitiendo unificar criterios y enfoques.



11

LOS CONTROLADORES

“

Son pequeñas porciones de **código**.

Su responsabilidad es **atender** a los distintos **request** del cliente y generar **comunicación** entre las **vistas** y los **modelos**.



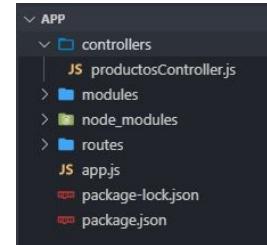
USTED ESTÁ AQUÍ



2

ARQUITECTURA DE ARCHIVOS

En la carpeta raíz del proyecto, crearemos la carpeta **controllers**. Adentro almacenaremos un controlador por cada recurso que tenga la aplicación. Cada controlador será un **módulo** que exportaremos, para luego requerirlo en donde lo necesitemos.



4

CREANDO UN CONTROLADOR

Crearemos un archivo para manejar nuestro recurso. Para nombrar los archivos se estila empezar con el **nombre** del **recurso** seguido de la palabra **Controller** usando el estilo **camelCase**.

```
JS productosController.js ×
controllers > JS productosController.js
1 // código
```

{}

5

CREANDO UN CONTROLADOR

Lo primero que hay que hacer es definir una variable en el archivo del controlador del recurso y asignarle un objeto literal.

```
const controlador = {};
```

Dentro del objeto, iremos definiendo los métodos que se encargarán de manejar, cada uno, un request en particular.

```
const controlador = {
  index: //mostrar listado de productos,
  show: //mostrar detalle de un producto,
  create: //enviar datos para agregar un producto,
};
```

6

CREANDO UN CONTROLADOR

Hasta el momento, la **lógica** de cómo manejar cada request que llegaba, la veníamos escribiendo en el archivo de rutas de cada recurso, en donde definíamos una **url** y un **callback** que se encargaba de manejar esa petición.

```
// código del archivo productos.js en la carpeta routes
router.get('/', (req, res) => {
  res.send('Index de productos');
});
```

Como **buena** práctica se suelen usar **nombres descriptivos** y **lógicos** para nombrar a los **métodos** de un **controlador**.



Para ir una **carpeta** hacia **atrás** cuando indicamos dónde está alojado un **archivo**, lo hacemos escribiendo lo siguiente al principio de la **ruta**:

```
.../
```



CREANDO UN CONTROLADOR

Al trabajar con el patrón MVC, podemos apoyarnos en sus reglas y dividir las responsabilidades de nuestros archivos. Siguiendo con esa línea, serán los **métodos** de cada controlador los que **recibirán** esos datos y **enviarán** la información correspondiente.

De modo que, quitaremos el callback que habíamos definido en las rutas y lo escribiremos en el método **index** del controlador de productos.

```
const controlador = {
  index: (req, res) => {
    res.send('Index de productos');
  }
};
```

IMPLEMENTAR UN CONTROLADOR

Primero hay que hacer visible todo el código que definimos en el controlador. Para eso, exportaremos la variable en la última línea del archivo.

```
module.exports = controlador;
```

Para empezar a usar los métodos que definimos, debemos requerir el módulo dentro del archivo de ruteo del recurso, en este caso `productos.js` tro de la carpeta `routes`

```
const productosController = require('../controllers/productosController');
```

IMPLEMENTAR UN CONTROLADOR

Con nuestro módulo ya visible en el archivo, es momento de terminar de configurar aquella ruta a la que le quitamos el callback.

La misma va a seguir recibiendo ese callback pero ahora no va a ser su responsabilidad definirlo. Para eso llamaremos al método **index** del controlador de productos y le pasaremos ese método como segundo parámetro. Al ser un callback no le escribimos los paréntesis.

```
router.get('/', productosController.index);
```

“

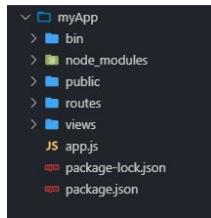
EXPRESS-GENERATOR

Express cuenta con un generador de proyectos llamado **express-generator**.



QUÉ TRAE EXPRESS-GENERATOR

Esta herramienta presenta una gran ventaja al momento de **empezar un proyecto** nuevo y es que, al instalarlo, trae consigo un **esqueleto** de carpetas, archivos y dependencias que nos puede servir para **inicializar** cualquier aplicación.



Este generador no trae consigo la carpeta **controllers** y sus **archivos**, por lo tanto tendremos que crearla nosotros si queremos respetar la arquitectura con la que venimos trabajando.



ESTRUCTURA DE ARCHIVOS

En la **carpeta raíz** encontramos el entry point `app.js`, y el archivo `package.json`.

Dentro de la **carpeta bin** encontramos el archivo `www` sin extensión. El mismo trae definida una lógica interna y se encargará de hacer que la aplicación corra.

Dentro de la **carpeta public** podremos guardar todos los recursos estáticos de nuestra aplicación.

Dentro de la **carpeta routes** estaremos administrando el route system de la aplicación. Encontramos los archivos `index.js` y `users.js`.

Dentro de la **carpeta views** encontramos dos vistas iniciales que trae el generador: `index.ejs` y `error.ejs`

INSTALANDO EXPRESS-GENERATOR

Para poder trabajar con el generador de proyectos, lo primero que hay que hacer es instalarlo globalmente para poder usarlo en el momento que queramos.

```
>_ npm install express-generator -g
```

Lo próximo será crear un proyecto Node usando Express con un comando que creará la carpeta del proyecto con el nombre que definamos nosotros. Adicionalmente, podemos configurar el motor de vistas que queremos usar. Si no lo aclaramos, por defecto se instalará el motor de vistas **pug**.

```
>_ express myApp --ejs
```

INSTALANDO EXPRESS-GENERATOR

Por último, dentro de la carpeta del proyecto, tenemos que correr el comando para instalar todas las dependencias que vinieron configuradas en el `package.json` para que el proyecto funcione.

```
>_ npm install
```

7

Echemos un pequeño **vistazo** a algunos de los **archivos** de la **estructura** que se generó.



app.js

En el primer **bloque** de este archivo encontramos los **requerimientos** de los **módulos** necesarios para empezar a desarrollar nuestra app. Entre ellos detectamos `express` (framework), `path` (trabajar con rutas de archivos), `http-errors` (manejar errores http).

```
var createError = require('http-errors');
var express = require('express');
var path = require('path');
var cookieParser = require('cookie-parser');
var logger = require('morgan');
var indexRouter = require('./routes/index');
var usersRouter = require('./routes/users');
var app = express();
```

11

MOTOR DE VISTAS

Los motores de vistas, también conocidos como motores de plantillas o template engines, nos permiten crear una estructura dinámica para las vistas de nuestro proyecto. Es decir, definir **bloques de contenido** que se pueden rellenar con **datos variables**.

Entre los más nombrados se encuentran EJS, Jade, Handlebars, entre otros.



8

index.js

Texto

10

QUÉ ES NODEMON

Esta herramienta nos permite **monitorizar** constantemente el **servidor** de Node.js.

Cada vez que queremos hacer modificaciones en el código de nuestro servidor, **Nodemon** se encargará de hacer los «stops» y «reloads» de nuestro servidor en Node.js, con lo que no tendremos que estar haciéndolo manualmente.



12

LEVANTAR EL SERVIDOR

Dentro de la carpeta del proyecto debemos correr el siguiente comando, en el cual indicaremos el archivo que queremos ejecutar.

```
>_ nodemon bin/www
```

Por último ingresar a <http://localhost:3000> para comprobar que el servidor se levantó correctamente.

INTRODUCCIÓN A TEMPLATE ENGINE

2.

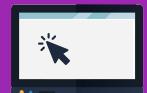
Dentro de esa estructura podremos **definir bloques** de contenido que se luego se irán rellenando con **datos variables**.

1.

Un *template engine* es un **motor** de **plantillas** que nos permite generar un archivo que contenga **estructuras dinámicas**.

3.

Dentro del **ecosistema** Express y el flujo **MVC**, el motor de plantillas es el encargado de **manejar** las **vistas**.



4.

Algunos motores de plantillas son:

- ★ **Handlebars**
- ★ **Mustache**
- ★ **EJS** (trabajaremos con éste)
- ★ **PUG**

5.

Debemos aclarar la extensión que corresponda –según el template que elijamos– para que nuestro archivo **html** se pueda transformar en un **archivo dinámico**.

INSTALACIÓN E IMPLEMENTACIÓN

Para instalar un **template engine** en nuestro proyecto lo haremos a través de la **terminal** usando el gestor de dependencias **NPM**.



USANDO UN TEMPLATE ENGINE

- Definir qué motor de plantilla queremos usar en nuestra aplicación, **en este caso ejs** y luego instalarlo con **npm**.

```
>_ npm i ejs --save
```

- Aclararle a Express cual es ese motor que vamos a estar utilizando a lo largo del proyecto utilizando el método `set()` con la propiedad 'view engine' y el valor del motor, **en este caso ejs**.

```
{ app.set('view engine', 'ejs');
```

3

LAS VISTAS

Al implementar un motor de plantillas, todas las vistas que creamos en nuestro proyecto deberán **almacenarse** en una **carpeta** específica.

Por defecto, Express va a ir a buscar esas vistas a una carpeta llamada **views**, que tendremos que crear en nuestro directorio.

Allí almacenaremos todos los archivos de vistas con la **extensión** que **corresponda** según el motor de plantillas que hayamos instalado.

En caso de querer almacenarlas en otra carpeta, deberemos aclararlo usando el método `set()`.

.set()

Es un **método** que nos permite definir **configuraciones** de express. Se ejecuta sobre la variable que tenga asociada la ejecución de express, en la mayoría de los casos llamada `app`.

Recibe dos **strings** como parámetros.

Para **configurar el motor de plantillas** le pasamos el string 'view engine' y el nombre del motor de plantillas que instalamos.

```
{ app.set('view engine', 'ejs');
```

Para **configurar la carpeta que almacena las vistas** le pasamos el string 'view' y la ruta absoluta hacia esa carpeta haciendo uso de `__dirname`.

```
{ app.set('view', __dirname + '/carpeta-vistas');
```

5

__dirname

Es una **variable** nativa de Node. Contiene la ruta exacta del archivo en el que nos encontramos cuando la llamamos.



Partiendo de la estructura de carpetas del ejemplo, si ejecutamos un `console.log` de `__dirname` desde `app.js` obtendremos el siguiente string:

```
>_ /Users/mi-proyecto
```

6

Para automatizar esta práctica y configurar todo en una sola línea podemos usar **express-generator** y ejecutar el comando

```
express --view=ejs carpetaProyecto
```

aclarando el *motor de plantillas* que instalaremos junto con el *nombre de la carpeta del proyecto.*



RENDERIZAR UNA VISTA

.render()

Es un **método** que se encuentra dentro del objeto **response** de la petición. Nos permite enviarle una vista al navegador para que este la renderice.

Recibe un **string** como parámetro: el nombre del archivo de la vista que queremos renderizar.

Es importante aclarar que, cuando le pasamos el nombre del archivo, **no hace falta aclararle la carpeta** en donde está almacenada esa vista -siempre y cuando hayamos configurado el template engine correctamente con el método `use()`.

Tampoco hace falta aclarar la **extensión** del archivo.

REPASO MVC

Las vistas se comunican con los **controladores** y reciben la información que ellos reciben de los **modelos**.

Para poder **renderizar** una **vista** es importante aclararle al **controlador** **qué vista** queremos enviar al navegador.



RECURSOS ESTÁTICOS

Son aquellos **recursos públicos** que manejamos dentro de nuestra aplicación: imágenes, archivos css, videos, archivos de javascript, etc.

Para poder disponer libremente de ellos en nuestro proyecto, hace falta aclararle a Express **dónde** vamos a estar almacenando esos recursos.

```
{}
app.use(express.static(__dirname + '/public'));
```

Con esa línea de código le estamos dando a Express acceso libre a todo lo que se encuentre dentro de la carpeta public.



El nombre de la carpeta puede ser el que queramos. Por convención se utiliza el nombre `'public'`.

REQUERIR UN RECURSO ESTÁTICO

Para acceder a alguno de estos recursos desde nuestros archivos, sólo hace falta aclarar la **ruta** hacia dicho recurso, comenzando la ruta siempre con una '/'.

```
html

```

¿Por qué?

Porque Express ya sabe que nuestros recursos estáticos se almacenan en la carpeta `public`. Por lo tanto, si bien no hace falta aclararlo en la ruta, tenemos que crear el path completo: `public/images/logo.png`

PASO A PASO

- Instalar ejs

```
npm i ejs --save
```

- Configurar ejs como el **template engine** de la app

```
app.set('view engine', 'ejs')
```

- Configurar el acceso a la carpeta de **recursos estáticos**

```
app.use(express.static(__dirname+ 'public'))
```



“

EJS trae consigo un conjunto de **etiquetas** que nos permiten integrar **funcionalidad** de *javaScript* dentro de nuestras estructuras *html*.



TAGS DE EJS

ESTRUCTURA BÁSICA

Para poder **implementar** las etiquetas que nos brinda el *template engine*, es necesario que nuestros archivos tengan la extensión `.ejs`.

Estos archivos seguirán soportando **todas** las etiquetas *html*, por lo tanto podremos desarrollar nuestras estructuras normalmente, sólo que ahora con la capacidad de añadirles **dinamismo**.



index.ejs

`<% ... %>`

Esta etiqueta nos permite incorporar código de *JavaScript*, como condicionales, estructuras de control de flujo, declaración de variables, etc.

¿Cómo la implementamos?

Por cada línea de *javascript* que escribamos, debemos encerrar ese código entre la etiqueta `<%` de apertura y la etiqueta `%>` de cierre. En el medio podemos escribir cualquier contenido *html*.

```
<% if(4 < 5) { %>
    <h2>El 4 es menor que el 5</h2>
<% } %>
```

3

4

{ código }

```
for(var i=0; i<5; i++){
}
```

Como primer paso podemos escribir la **sentencia de JavaScript** que queremos **embeber** en nuestra estructura general.

{ código }

```
<% for(var i=0; i<5; i++) { %>
<% } %>
```

Luego, **encerrar** cada línea entre las etiquetas de cierre y apertura de `ejs` que nos permiten embeber código de *javascript*: `<%` y `%>`

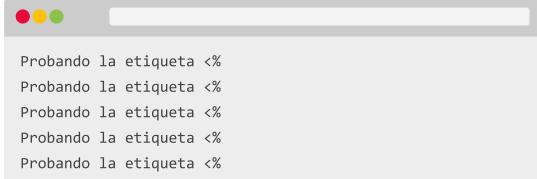
5

6

{ código }

```
<% for(var i=0; i<5; i++) { %>
    <p> Probando la etiqueta <% </p>
<% } %>
```

Finalmente, escribiremos el **contenido** que queremos mostrar por cada iteración del ciclo for declarado.
En esta instancia, podemos incorporar las etiquetas de html que queramos.



{}

7

<%= ... %>

Esta etiqueta nos permite **imprimir** un **valor dinámico** y de esta manera incorporarlo en la estructura html general.

¿Cómo la implementamos?

Al valor dinámico que busquemos renderizar, los tendremos que encerrar entre la etiqueta <%= de apertura y la etiqueta %> de cierre.

```
<% if(nombre) { %>
    <h2>¡Hola <%= nombre %>! </h2>
<% } %>
```

{ código }

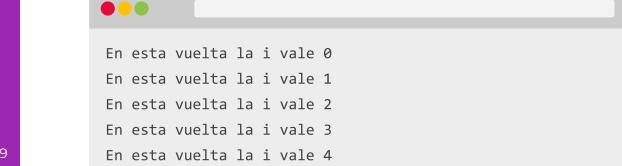
```
<% for(var i=0; i<5; i++) { %>
    <p> En esta vuelta la i vale <%= i %> </p>
<% } %>
```

En este ejemplo, buscamos **imprimir** el valor de la variable i en cada iteración.

{ código }

```
<% for(var i=0; i<5; i++) { %>
    <p> En esta vuelta la i vale <%= i %> </p>
<% } %>
```

A esas etiquetas de ejes las podemos rodear de las etiquetas de html que queramos. De esta forma, incorporamos el contenido dinámico dentro de la estructura html general.



9

10

<%

La usamos para embeber código de JavaScript como **condicionales**, **estructuras de control** de flujo, etc.



<%=

La usamos para embeber código e **imprimir** el **resultado** de una expresión o un valor.

“

PARÁMETROS COMPARTIDOS

La gran **ventaja** de utilizar un motor de plantillas, es que nos permite **compartir** información **desde los controladores** hacia las **vistas** y disponer de ella como queramos.



¿CÓMO MUESTRO UNA VISTA?

Sobre el objeto **response** ejecutamos el método `.render()`, pasándole como argumento el nombre de la vista que queremos renderizar.

```
const controller = {
  mostrarPeliculas: (req, res) => {
    res.render('peliculas')
  }
}
```

`.render()`

El método `.render()` puede recibir un **objeto literal** como segundo parámetro. Este objeto tendrá almacenada la información que queremos enviar en conjunto con la vista que estemos renderizando.

```
const controller = {
  mostrarPeliculas: (req, res) => {
    res.render('peliculas', {})
  }
}
```

3

4

EL OBJETO COMO PARÁMETRO

Al objeto tendremos que asignarle una *propiedad* y un *valor*.

- El **valor** será la **información** que queremos que viaje hasta la vista.
- El **nombre** de la **propiedad** será el que usaremos para disponer de esa información dentro del archivo de la vista.

```
const peliculas = ['Deadpool', 'The Joker', 'Batman'];

const controller = {
  mostrarPeliculas: (req, res) => {
    res.render('peliculas', {listaPeliculas: peliculas})
  }
}
```

Podemos asignarle la cantidad de **propiedades** que **deseemos** a ese objeto.

La vista, por su parte, no recibirá un **objeto literal** con todas esas propiedad sino que recibirá las propiedades **sueltas** como **variables**.



5

EL OBJETO COMO PARÁMETRO

```
const peliculas = ['Deadpool', 'The Joker', 'Batman'];

const controller = {
    mostrarPeliculas: (req, res) => {
        res.render('peliculas', {listaPeliculas: peliculas,
                                extranjeras: true,
                                genero: 'superhéroes' })
    }
}
```

Cada una de esas propiedades será una **variable** que tendremos disponible en la **vista** y así disponer de la información que almacena cada una.

MOSTRAR LA INFORMACIÓN

Para **mostrar** la información en la vista, haremos uso de los **tags** que nos provee el motor de plantillas ejs y llamaremos a la **propiedad** que creamos en el objeto para almacenar la información.

```
{% for(let pelicula of listaPeliculas) { %}
    <h2>El título de la película es <%= pelicula %> </h2>
{% } %}
```

7

Propiedad del objeto que viajó a la vista con la **información** desde el controlador

Tag para **embeber** código JS

Tag para embeber código e **imprimir** un valor

8

“

PLANTILLAS PERSONALIZADAS

Existen maneras de **modularizar** nuestra **estructura** html logrando así simplificar mucho el **desarrollo** y **ensamblado** final de nuestro proyecto.



CÓMO MODULARIZAR

Lo primero a tener en cuenta es detectar aquellas porciones de código que vemos repetidas en todos los archivos del proyecto. En una estructura corriente de html podemos encontrar que todos los archivos fácilmente van a contar con: un head, una barra de navegación y un footer, por mencionar algunas. Cada una de estas partes, se convertirá en un **archivo nuevo** que contendrá **únicamente esa porción** de código.

Si estamos trabajando con un **template engine** deberemos almacenar estos archivos dentro de la carpeta **views** -o la que hayamos configurado por defecto- para no cortar el flujo de trabajo.

PARTIENDO EL CÓDIGO

Tendremos que **cortar** el código que queramos modularizar y **pegarlo** en un **nuevo archivo** al cual le asignaremos un nombre que represente la parte que contiene, aclarando siempre la extensión del motor de plantillas que usemos, por ejemplo: **head.ejs**.

```
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="/css/master.css">
</head>
```



Como buena práctica se sugiere almacenar estos archivos dentro de una carpeta llamada **partials**.

REUTILIZANDO EL CÓDIGO

Para disponer del código que modularizamos, necesitamos hacer uso de la función **include()** que nos provee ejs. La misma recibe como parámetro un **string** que será la **ruta** hacia el **archivo** que queremos incluir.

```
html1 include('./partials/head')
```

Por último, deberemos escribir la etiqueta **<%-**, que imprimirá en el documento el contenido exacto que retorne el **include()**.

```
html2 <!DOCTYPE html>
<html lang="en">
<%- include('./partials/head') %>
<-- Código -->
```

¿POR QUÉ MODULARIZAR?

Porque nos permite **ordenar** nuestro código, ser menos **repetitivos**, y conseguir un mejor y más simple **mantenimiento** del código.



“

El protocolo **HTTP** define una serie de **reglas** que es necesario seguir para que la información pueda ser **procesada**.



MÉTODOS HTTP

MÉTODOS DE TRANSACCIÓN

El flujo de **transacción** que propone el protocolo HTTP viene acompañado de varios **métodos** que **definen** lo que pasará con cada pedido que le solicite un cliente a un servidor.

En **express** podemos implementar los métodos sobre la ejecución de **express - app** - o sobre el sistema de rutas del sistema - **router**.



Como buena práctica se sugiere trabajar los métodos directamente sobre la ejecución de **express** entendiendo que es la arquitectura más recomendada.

MÉTODO GET

Con este método podemos **solicitarle** datos al servidor.



Al acceder a una página a través de una **url** estoy haciendo una petición con **get**.

3

MÉTODO POST

Con este método podemos **enviarle** datos al servidor.



Al registrarme en un sitio nuevo con mis datos estoy haciendo una petición con **post**.

DIFERENCIAS ENTRE GET Y POST

Los pedidos por GET...	Los pedidos por POST...
vian por url y queda visible la información	vian ocultos y queda encriptada la información
pueden ser cacheados	no pueden ser cacheados
pueden guardarse en marcadores	no pueden guardarse en marcadores
tienen restricción en la longitud	no tienen restricción en la longitud
se usan para manipular datos que no sean sensibles	son más seguros y se usan para manipular datos sensibles
no se reenvían al recargar un sitio	se reenvían al recargar un sitio

MÉTODO **PUT**

Con este método podemos **reemplazar** información existente.

*Al cambiar información personal en instagram estoy haciendo una petición con **put**.*



MÉTODO **DELETE**

Con este método podemos **borrar** un registro existente en el servidor.

*Al borrar una foto de facebook estoy haciendo una petición con **delete**.*



No todos los navegadores implementan **PUT**, **PATCH** y **DELETE** con lo cual varios frameworks tienen algunas prestaciones para simularlos.



“

RUTAS PARAMETRIZADAS



Este tipo de rutas nos permite terminar de **configurar** nuestra **estructura** HTML para que sea 100% **dinámica**.

COMPORTAMIENTO DINÁMICO

Para lograr un comportamiento dinámico en las rutas que definimos en nuestro proyecto, es necesario entender que, al momento de hacer una petición al servidor -sin importar qué método sea-, podemos enviar **parámetros** en la **url**.

¿Qué quiere decir?

Que podremos definir qué valor de la url irá **variando** en cada petición.

3

Podemos definir la **cantidad** de **parámetros** dinámicos que queramos dentro de una ruta, *sin importar el orden en que lo hagamos*:

```
router.get("/:genero/:region/:anio", (req, res) ...)
```



Cuando a la ruta **no le llegan** todos los parámetros que tiene definidos, la misma **no funcionará**.

PARÁMETROS DINÁMICOS

Para definir un parámetro dinámico en la ruta escribimos el carácter `:` seguido de un **nombre representativo** del valor que estará viajando por url.

Suponiendo que estamos en el archivo de rutas del recurso películas, podríamos definir una ruta parametrizada así:

```
{}
  router.get("/:genero", peliculasController... )
```

De esta forma creamos una ruta que, según el género que reciba, devolverá una respuesta distinta en cada caso:

- `https://localhost/peliculas/drama` devuelve sólo las de drama
- `https://localhost/peliculas/accion` devuelve sólo las de acción

4

PARÁMETROS OPTATIVOS

Para definir un parámetro **optativo** en la ruta escribimos el carácter `:` seguido de un **nombre representativo** del valor que estará viajando por url, seguido del carácter `?` al final.

Suponiendo que estamos en el archivo de rutas del recurso películas, podríamos definir un parámetro optativo así:

```
{}
  router.get("/:duracion?", peliculasController... )
```

Si la ruta recibe ese parámetro, filtrará la respuesta según ese valor. Si no llegara a recibir el parámetro, seguirá funcionando:

- `https://localhost/peliculas/120` devuelve sólo las de esa duración
- `https://localhost/peliculas` devuelve todas las películas

El **tipo** de parámetros y la **cantidad** que definamos en nuestras rutas va a depender **directamente** de la **lógica** de nuestra aplicación y de lo que necesitemos **mostrar** en cada caso.



ATRAPANDO VALORES

A través de la propiedad `params` del objeto `request` podremos acceder a los valores que viajan en la url.

`params` es un objeto literal con clave:valor:

- la clave será el **nombre del parámetro** que definimos al crear la ruta
- el valor será el dato que viaje en la url

Teniendo una ruta parametrizada que usa el método `genero` del controlador de películas...

```
{}
router.get("/:genero", peliculasController.genero);
```

... veamos cómo atrapar ese género que llegará por url

ATRAPANDO VALORES

Dentro del método `genero` usaremos el objeto `request` para acceder al objeto `params` y mostrar por consola el resultado:

```
const controller = {
  genero: (req, res) => {
    console.log(req.params.genero)
  }
}
```

Si la url fuera: `https://localhost/peliculas/terror`, veríamos por consola:

```
>_
  terror
```

Recordá que `params` es un **objeto literal** al cual podrás acceder con la notación de punto a todas sus **propiedades y valores**.

*Dentro de los **métodos** de tus rutas irás desarrollando la **lógica** necesaria en cada caso para trabajar de manera **dinámica** con cada parámetro que te llegue.*



“

PROCESAMIENTO GET

Las peticiones que se hacen por **get** son todas aquellas que vienen **directamente** desde la **URL** del navegador o **internamente** en la página desde un **enlace**.



MANEJAR PETICIONES GET

Comúnmente usamos el método get para:

- Retornar **vistas**
- Retornar **archivos**
- Retornar **datos**

Cuando definimos una ruta podemos hacerlo directamente sobre la **ejecución de express**, implementar un **sistema de ruteo** o también **incorporar controladores** que se encarguen de manejar las rutas.

Sin importar el camino que elijamos para implementar en nuestra aplicación, es en el **callback de la ruta que estamos definiendo** en donde escribiremos la lógica para manejar la petición que esté llegando.

MANEJAR PETICIONES GET

```
// enrutador
router.get("/peliculas", (req, res) => { res.render('peliculas') });

// enrutador con controlador
router.get("/peliculas", peliculasController.todas);
... const controller = {
  ...todas: (req, res) => { res.render('peliculas') }
};

// sobre la ejecución de express
app.get("/peliculas", (req, res) => { res.render('peliculas') });
```

3

4

MANEJAR PETICIONES GET

```
router.get("/peliculas", (req, res) => { res.render('peliculas') });

router.get("/peliculas", peliculasController.todas);
... const controller = {
  ...todas: (req, res) => { res.render('peliculas') }
};

app.get("/peliculas", (req, res) => { res.render('peliculas') });
```

CALLBACK

5

QUERY STRING

Es una **cadena de texto** -conocida como **cadena de consulta**- que viaja en la **url** al momento de hacer una **petición** al servidor.

El **query string** comienza al final de la ruta con el signo **? .** Está formado por el par **clave:valor**. En el caso de haber más de un par son separados por el carácter **&** :

https://www.youtube.com/results?search_query=digital+house

QUERY STRING

?search_query=digital+house

Inicio de la query

Clave de la query

Operador de asignación

Valor de la clave

6

Para **acceder** al *query string* dentro del **callback** que maneja la petición lo haremos a través de la propiedad **query** del objeto **request**.

Esta propiedad es un **objeto literal**, en donde sus **claves** y **valores** serán las mismas que viajen en la url:

```
console.log(req.query.search_query)  
// digital house
```



PROCESAMIENTO POST

MANEJAR PETICIONES POST

Comúnmente usamos el método post para:

- Enviar información sensible al servidor
- Crear un nuevo recurso

Cuando definimos una ruta podemos hacerlo directamente sobre la ejecución de express, implementar un sistema de ruteo o también incorporar controladores que se encarguen de manejar las rutas.

Sin importar el camino que elijamos para implementar en nuestra aplicación, es en el callback de la ruta que estamos definiendo en donde escribiremos la lógica para manejar la petición que esté llegando.

MANEJAR PETICIONES POST

En un contexto en donde quisiéramos agregar una nueva película a nuestro sistema, tendríamos que crear dos rutas: una que muestre el formulario de creación y otra que se encargue de procesar la información.

```
// ruta que envía un formulario a la vista → GET
router.get('/película/crear', (req,res) => {res.render('crear')});

// ruta que procesa la información del formulario → POST
router.post('/película/crear', (req,res) => {...});
```

Los nombres de las rutas pueden ser iguales porque cada una está implementando un método diferente.

CONFIGURAR EL FORMULARIO

Las peticiones que se hacen por post son todas aquellas que viajan a través de un formulario. Es necesario que el mismo tenga seteados los atributos:

- method → donde escribiremos el método HTTP que usaremos para enviar la información
- action → donde escribiremos la ruta a donde viajará esa información para ser procesada

```
<form method="POST" action="/película/crear">
  ...
</form>
```

CAPTURAR LA INFORMACIÓN

Para poder trabajar con los datos que se envían desde el formulario, es necesario configurar el entorno de nuestra aplicación para que sea capaz de capturar esa información.

Si estamos trabajando con express-generator, esta configuración se creará por defecto durante la instalación.

De lo contrario, tendremos que agregar estas dos líneas de código en el archivo app.js :

```
app.use(express.urlencoded({ extended: false }));
app.use(express.json());
```

De esta forma le estamos **aclarando** a la aplicación que todo aquello que **llegue** desde un **formulario**, queremos capturarlo en forma de **objeto literal**.

Y a su vez, tener la posibilidad de **convertir** esa **información** en un formato **json**, en caso de necesitarlo.



req.body

En el **request** de la petición encontramos la propiedad `body`, un objeto literal que contendrá **toda** la información del formulario:

- El nombre de cada **clave** de ese objeto, será el **nombre** del atributo `name` de cada input del formulario
- El **valor** será el dato que se haya ingresado en ese campo

```
html
<form method="POST" action="/pelicula/crear">
  Título: <input type="text" name="titulo" value="">
  ...
</form>
```

```
{}
router.post('/pelicula/crear', (req,res) => {
  console.log(req.body) // { titulo: Batman }
});
```

Para **cerrar** el ciclo del **request** y **response** que hace el servidor es necesario hacer un **redireccionamiento** -después de implementada la lógica- usando el método `redirect()` sobre el **response**:

```
res.redirect('/peliculas');
```



“

PROCESAMIENTO PUT Y DELETE

La **información** que viaje como petición **PUT** y **DELETE** debe hacerlo a través de un **formulario**.



1.

PETICIONES PUT

MANEJAR PETICIONES PUT

Usamos el método put para:

- Enviar información sensible al servidor de manera segura
- Modificar un recurso existente

Cuando definimos una ruta podemos hacerlo directamente sobre la ejecución de express, implementar un sistema de ruteo o también incorporar controladores que se encarguen de manejar las rutas.

Sin importar el camino que elijamos para implementar en nuestra aplicación, es en el callback de la ruta que estamos definiendo en donde escribiremos la lógica para manejar la petición que esté llegando.

4

MANEJAR PETICIONES PUT

En un contexto en donde quisiéramos **modificar** datos de una película almacenada en nuestro sistema, tendríamos que crear dos **rutas**: una que **muestre** el formulario de **edición** y otra que se encargue de **procesar** la información.

En ambas rutas, deberemos definir un parámetro que nos ayude a identificar al **recurso único** que estamos queriendo modificar. Generalmente, se usa el **id**.

```
// ruta que envía un formulario de edición a la vista → GET
router.get('/pelicula/:id/editar', (req,res) => {res.render('editar')});

// ruta que procesa la información del formulario → PUT
router.put('/pelicula/:id/editar/', (req,res) => {...});
```

{}

5

2.

PETICIONES DELETE

MANEJAR PETICIONES DELETE

Usamos el método delete para:

- Eliminar un recurso existente

Cuando definimos una ruta podemos hacerlo directamente sobre la ejecución de express, implementar un sistema de ruteo o también incorporar controladores que se encarguen de manejar las rutas.

Sin importar el camino que elijamos para implementar en nuestra aplicación, es en el callback de la ruta que estamos definiendo en donde escribiremos la lógica para manejar la petición que está llegando.

7

MANEJAR PETICIONES DELETE

En un contexto en donde quisiéramos eliminar una película almacenada en nuestro sistema, tendríamos que crear una ruta que se encargue de buscar ese recurso y eliminarlo. En la ruta deberemos definir un parámetro que nos ayude a identificar al recurso que estamos queriendo eliminar. Generalmente, se usa el id.

```
{}
// ruta que procesa la información del formulario → DELETE
router.delete('/pelicula/:id/eliminar', (req,res) => {...});
```

8

En el callback de las peticiones **DELETE** y **PUT** tendremos que definir la **lógica** de lo que queremos implementar con esa información que recibimos.

Después, haremos un **redireccionamiento** para cortar el ciclo *request/response*.

```
res.redirect('/peliculas');
```



CONFIGURAR EL FORMULARIO

Para terminar de habilitar el envío de información a través de alguno de estos dos métodos, tenemos que agregarle un **query string** al **action** del formulario:

- ?_method=PUT
- ?_method=DELETE

De esta forma estamos aclarando que, sin importar el método original que tenga seteado el formulario, queremos enviar la información usando PUT o DELETE.

```
html
<form method="POST" action="/pelicula/:id/editar?_method=PUT">
  ...
</form>
```

HABILITAR MÉTODOS HTTP

Los métodos **PUT** y **DELETE** no son soportados por todos los navegadores. Para poder implementarlos en nuestro formulario es necesario instalar el paquete **method-override**:

```
>_
npm install method-override --save
```

Una vez instalada, hay que configurar la aplicación en **app.js** para poder **sobreescribir** el método original y poder implementar los métodos **PUT** o **DELETE**:

```
{}
const methodOverride = require('method-override');
app.use(methodOverride('_method'));
```

11

Aunque los formularios sigan teniendo configurado el método POST, la información estará viajando a través del **método** que **definamos** en cada **ruta**.

De esta forma, es fácil identificar **qué hace** cada ruta de nuestro sistema tan solo viendo el **método** que **implementa**.



“



Es un error que **envía el servidor** cuando **no encuentra** un recurso que solicitó el **cliente**.

ERROR 404

PREPARANDO EL SISTEMA

Todo sistema debe estar preparado para manejar las solicitudes erradas. Express trae consigo una respuesta automatizada para cuando el usuario **solicita** un contenido que ya **no existe** en el servidor. En el entry point de la aplicación -la mayoría de las veces `app.js`- podemos configurar esta respuesta implementando el método `use()` sobre la ejecución de express.

```
{}
app.use((req, res, next) => {
  ...
})
```

`use()` va a recibir un callback con 3 argumentos: un **request**, un **response** y el **paso siguiente** a ejecutar después de que se haya ejecutado el callback.

PREPARANDO EL SISTEMA

Dentro de este método, definiremos cómo va a reaccionar el sistema cada vez que el usuario quiera acceder a una ruta que no existe. Sobre el `response` implementaremos el método `status()` seguido del método `render()` para poder renderizar la vista que verá el usuario cada vez que se presente este escenario:

```
{}
app.use((req, res, next) => {
  res.status(404).render('not-found');
})
```

INTRODUCCIÓN A PATH

2.

Para atajar esas diferencias, existe el **paquete nativo path** que trae consigo herramientas que nos **facilitan** el trabajo con **rutas** de archivos y directorios.

1.

Según el **sistema operativo**, las rutas se van a ver de diferente manera:

- ★ **LINUX y MAC**
/users/dh/saludo.txt
- ★ **WINDOWS**
C:\users\dh\saludo.txt

3.

Para trabajar con él hay que **requerir** el módulo y guardarla en una variable dentro del archivo en el que queremos implementarlo.

```
const path = require('path');
```

4.

path.join()

Método que nos permite **describir** de manera sencilla la **ruta** de una carpeta o un archivo. *Une los parámetros que recibe y devuelve una única ruta.*

```
path.join('users','dh','saludo.txt'));  
// users/dh/saludo.txt
```

5.

path.dirname()

Método que nos permite **conocer la carpeta** en la que se **encuentra** un archivo. *Recibe la ruta del archivo y devuelve la ruta de la carpeta que lo contiene.*

```
path.dirname('users/dh/test.txt');  
// user/dh
```

6.

path.extname()

Método que nos permite **conocer** la **extensión** de un archivo. *Recibe la ruta del archivo y devuelve su extensión.*

```
path.extname('users/dh/saludo.txt');  
// .txt
```

INTRODUCCIÓN A FILE SYSTEM

1.

Es un **paquete nativo** de Node que trae consigo métodos que **facilitan** la **lectura** y **escritura** de **archivos** y **carpetas** de nuestro sistema operativo.

2.

Para trabajar con él hay que **requerir** el módulo y guardarla en una variable dentro del archivo en el que queremos implementarlo.

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

3.

A través de este **paquete** vamos a poder:

- ★ **crear** carpetas y archivos
- ★ **borrar** carpetas y archivos
- ★ **leer** archivos
- ★ **escribir** archivos
- ★ **cambiar** permisos y más

ESCRITURA DE ARCHIVOS

PREPARANDO EL CONTENIDO

Los **métodos** de **escritura** de archivos que trae `file system` sólo pueden recibir contenido que sea de tipo `string`.

Por eso, es importante convertir a texto todo aquello que queramos escribir en un archivo. Para convertir un objeto podemos usar el método `JSON.stringify()`.

```
const fs = require('fs');
let pelicula = {titulo:'Titanic', minutos:500};
let peliculaJson = JSON.stringify(pelicula);
fs.writeFileSync('lista-colores.txt', peliculaJson);
```

.writeFileSync()

Es un método que trae el paquete nativo `file system` que nos permite escribir archivos. Recibe dos parámetros:

- El primero, el **archivo** en donde queremos escribir
- El segundo, el **contenido** que queremos escribir

Algo importante a tener en cuenta es que, si le pasamos el nombre de un archivo que aún no existe, el mismo método se encargará de crearlo.

```
{} const fs = require('fs');
fs.writeFileSync('estrenos-2020.txt','Titanic 2');
```

INFORMACIÓN IMPORTANTE

- ★ Cuando los datos que queremos escribir en un archivo nos llegan desde un **formulario**, capturarlos con la propiedad `body` del objeto `request`.
- ★ Usar el método `appendFileSync()` cuando queremos **agregar** datos en un archivo existente.
- ★ `writeFile()` se ejecuta de manera **asincrónica**, es decir que **no bloquea** la ejecución del resto del código.
- ★ `writeFileSync()` se ejecuta de manera **sincrónica**, es decir que **bloquea** la ejecución del resto del código hasta que termine con la operación.

LECTURA DE ARCHIVOS

Si estamos leyendo un archivo **JSON**, hay que convertir ese string en un **objeto literal** para poder manipular los datos usando el método **JSON.parse()**.

```
let users = fs.readFileSync('users.json',{encoding: 'utf-8'});
let usersJson = JSON.parse(users);
```



.readFileSync()

Es un método que trae el paquete nativo `file system` que nos permite **recuperar datos** de un archivo para poder leerlos.

Como **primer parámetro** recibe la **ruta** del archivo que queremos leer:

```
{}
const fs = require('fs');
let sitcoms = fs.readFileSync('sitcoms.txt');
```

Para poder decodificar los datos que el método devuelve, es **fundamental** pasarle un **segundo parámetro** aclarando el tipo de encoding:

```
{}
let sitcoms = fs.readFileSync('sitcoms.txt',{encoding: 'utf-8'});
```

INFORMACIÓN IMPORTANTE

- ★ **readFile()** se ejecuta de manera **asincrónica**, es decir que **no bloquea** la ejecución del resto del código.
- ★ **readFileSync()** se ejecuta de manera **sincrónica**, es decir que **bloquea** la ejecución del resto del código hasta que termine con la operación.

“

Cuando trabajamos con **datos sensibles** es fundamental almacenarlos **encriptados**, para **preservar** la información en caso de que un tercero acceda a ella.



HASHING

QUÉ ES UN HASH

En informática, las funciones de hasdeo nos permiten **encriptar** datos. Es decir, **transformar** un texto plano en una nueva serie de caracteres -con una longitud fija- imposible de descifrar para el ojo humano.

Es por eso que estas funciones vienen acompañadas de dos características principales:

- la opción de **encriptar un dato**
- la opción de **comparar** un dato entrante con un dato **hasheado** para verificar si coinciden o no

El paquete `bcrypt` nos permite incorporar estas funciones en nuestro proyecto de Node.

Para usarlo hay que instalarlo a través de npm.

```
npm install bcrypt --save
```



.hashSync()

Es un **método** que trae el paquete `bcrypt` que nos va a permitir encriptar datos. Recibe dos parámetros:

- El **dato** que queremos encriptar
- La **sal** que le queremos añadir a la encriptación

¿Qué es la sal?

Un pequeño dato añadido que hace que los hash sean significativamente más difíciles de crackear. En este contexto se le suele pasar 10 o 12.

```
const bcrypt = require('bcrypt');
let passEncriptada = bcrypt.hashSync('monito123', 10);
```

.compareSync()

Es un **método** que trae el paquete `bcrypt` que nos va a permitir **comparar** un texto plano contra un hash para saber si coinciden o no. Este método **retorna** un **booleano** y recibe dos parámetros:

- El primero, el **texto plano**
- El segundo, el **hash** con el que lo queremos comparar

```
let check = bcrypt.compareSync('monito123', passEncriptada);
console.log(check); // true
```

{}

INFORMACIÓN IMPORTANTE

- ★ **hash()** y **compare()** se ejecutan de manera **asincrónica**, es decir que **no bloquea** la ejecución del resto del código.
- ★ **hashSync()** y **compareSync()** se ejecutan de manera **sincrónica**, es decir que **bloquea** la ejecución del resto del código hasta que termine con la operación.

“

SUBIDA DE ARCHIVOS

Cuando queremos subir **archivos** a través de un formulario para **almacenar** en un servidor, se requiere de un **proceso** específico para poder lograrlo.



CONFIGURAR EL FORMULARIO

El primer paso para **subir** un archivo es **configurar** el formulario en la estructura html para que tenga todo lo necesario para que el mismo viaje hasta el servidor. Esos requerimientos son:

- Agregar el **input** de tipo **file** que le permite al usuario subir un archivo
- Agregar en la etiqueta **form** el **atributo** **enctype="multipart/form-data"**

```
html
<form method="POST" action="" enctype="multipart/form-data">
  <input type="file" name="avatar">
  ...
</form>
```

El paquete **multer** trae consigo funcionalidades que nos van a permitir procesar y almacenar esta subida de archivos.

Para usarlo hay que instalarlo a través de npm.

```
npm install multer --save
```



.diskStorage()

Es un **método** que trae el paquete **multer** que nos va a permitir operar con los archivos que estemos queriendo procesar. Recibe como parámetro un **objeto literal** con dos propiedades:

- **destination**, donde definiremos la carpeta donde se va a almacenar el archivo
- **filename**, donde indicaremos con qué nombre se guardará ese archivo en el servidor

Ambas propiedades son funciones. Cada una de ellas recibe tres parámetros: el **request**, el **archivo** y un **callback**.

{ código }

```
var storage = multer diskStorage({
  destination: (req, file, cb) => {
    ...
  },
  filename: (req, file, cb) => {
    ...
  }
})
```

destination

En **destination**, usaremos el callback para definir la carpeta en donde queremos almacenar los archivos. El primer parámetro será `null`, el segundo, la **ruta** hacia la carpeta de destino.

```
{ } var storage = multer.diskStorage({  
  destination: (req, file, cb) => {  
    cb(null, 'public/img/avatars')  
  },  
  filename: (req, file, cb) => {  
    ...  
  }  
})
```

7

filename

En **filename**, usaremos el callback para definir el nombre con el que guardaremos el archivo. El primer parámetro será `null`, el segundo, la **nombre**. Aquí usaremos la variable `file` junto con el paquete `path`.

Para crear el nombre del archivo, usaremos el método `extname()` del paquete, pasándole como parámetro el nombre original del archivo para que nos devuelva únicamente su extensión.

```
{ } file.fieldname + '-' + Date.now() + path.extname(file.originalname)
```

En la [documentación oficial de la librería](#), podemos encontrar toda esta **configuración** ya definida para copiar e incorporar a nuestro proyecto.



{ código }

```
{ } var storage = multer.diskStorage({  
  destination: (req, file, cb) => {  
    cb(null, 'public/img/avatars')  
  },  
  filename: (req, file, cb) => {  
    cb(null, file.fieldname + '-' + Date.now() + path.extname(file.originalname))  
  }  
})  
var upload = multer({ storage: storage });
```



Deberemos incorporar este código en **cada archivo** en donde exista una ruta que esté implementando la subida de archivos.

CONFIGURAR LAS RUTAS

A la ruta que se encargue de manejar la petición de la subida de archivos, habrá que pasarle un **nuevo parámetro** antes del callback: el método `upload.any()`.

```
{ } app.post('/', upload.any(), (req, res) => {  
  ...  
})
```

Adicionalmente, habrá que pasarle un **tercer parámetro** al callback -comúnmente nombrado `next`- para que todo funcione.

```
{ } app.post('/', upload.any(), (req, res, next) => {  
  ...  
})
```

PASO A PASO

- Agregar un `<input type="file">` en el formulario
- Agregar el atributo `enctype="multipart/form-data"` en la etiqueta `<form>`
- Instalar **multer**
- Requerir el módulo e implementar la configuración del método `diskStorage()` -extraída de la documentación oficial de multer- en los archivos de rutas que lo requieran
- En la propiedad `destination`, configurar la **ruta** de la carpeta en donde se almacenarán los archivos

11

8

10

12

PASO A PASO

- En la propiedad `filename`, configurar el **nombre** con el que se guardará el archivo
 - ◆ Usar `path.extname(file.originalname)` para obtener la extensión del archivo
- Agregarle `upload.any()` como segundo parámetro -antes del callback- a todas las rutas que implementen subida de archivos
- Agregar la variable `next` como tercer parámetro al callback

A cada **ruta** dentro de la aplicación que esté **procesando archivos**, deberemos configurarle estos parámetros para que implemente toda la **funcionalidad** de multer.



INTRODUCCIÓN A MIDDLEWARES

2.

Toda petición requiere de una serie de pasos para ser procesada. **Express** divide esas responsabilidades en **funciones** denominadas **middlewares**.

4.

Las rutas que requieran middlewares -además del *request* y el *response*- deberán recibir un tercer parámetro: la función **next**.

1.

Express trata al **request** y **response** como *objetos*. Recibe una petición, la *procesa* y devuelve un **objeto** como **respuesta**.

3.

Los middlewares son **funciones** que se ejecutan en medio de la ejecución del pedido del cliente y la respuesta del mismo.

5.

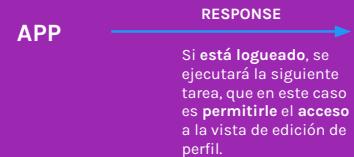
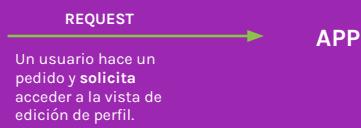
next permite la ejecución en cadena de *todas las funciones necesarias* que deberán ejecutarse **antes** de enviarle la respuesta al cliente.

6.

Con los *middlewares* podemos **aislar** código que nos permita resolver un escenario en particular e **implementarlo** en donde sea necesario, evitando así *repetir código*.

7.

Estas funciones agregan una capa de **seguridad**. Si la función que definimos como middleware *no retorna* lo esperado, la misma **cortará** la ejecución en cadena y devolverá la respuesta seteada para esos casos.





“

MIDDLEWARES A NIVEL APPLICACIÓN

Son aquellos middlewares que queremos que se ejecuten **siempre a lo largo de toda la aplicación**, sin importar a qué ruta ingrese el usuario.



CÓMO CONFIGURARLO

Invocando el método `app.use()` estamos configurando un middleware que se va a implementar en toda la aplicación. Este método recibe un callback con tres parámetros:

- el objeto `request`
- el objeto `response`
- la función `next`

```
{  
  app.use(function(req, res, next) {  
    ...  
  })  
}
```

Algunos de los middlewares a **nivel aplicación** que ya venimos utilizando son:

```
// configuración de carpeta de archivos estáticos  
app.use(express.static(__dirname + '/public'));  
  
// configuración de ruteo  
const rutasProductos = require('../routes/products');  
app.use('/', rutasProductos);
```



3

QUÉ ES NEXT

`next` es un callback que se va a encargar de apilar todos los middlewares que apliquen a una misma petición, y **ejecutarlos** uno tras otro. Cuando llegue al último y, si se ejecutaron correctamente, pasará al siguiente paso que es llegar al controlador que maneja esa ruta.

Por eso siempre al terminar cada middleware, ejecutamos `next`.

```
{  
  app.use(function(req, res, next) {  
    ...  
    next();  
  })  
}
```

Configuremos un middleware que maneje los errores **404** de la aplicación.



5

{ código }

```
app.use((req,res, next) => {  
  res.status(404).render('404-page');  
  next();  
});
```

{ código }

```
app.use();
```

En el **entry point** de la aplicación, llamo al método `use()` para configurar un middleware que aplique a todas las peticiones del sitio.

7

8

{ código }

```
app.use((req,res, next) => {  
  ...  
});
```

{ código }

```
app.use((req,res, next) => {  
  res.status(404).render('404-page');  
});
```

Si la página **no existe**, devolverá un error de status 404 y, adicionalmente, renderizará la vista que tenga diseñada para ese escenario.

9

10

{ código }

```
app.use((req,res, next) => {  
  res.status(404).render('404-page');  
  next();  
});
```

La función `next` se encargará de ejecutar el próximo paso. Si la página **existe**, llamará al controlador y este devolverá la vista solicitada.

11

MIDDLEWARES A NIVEL RUTAS

Son aquellos middlewares que se **aplicarán** únicamente a la **rutas** en donde los **definamos**.



CÓMO CONFIGURARLO

Ya sea que estemos utilizando un sistema de ruteo, o rutas definidas directamente sobre la ejecución de express, para aplicarles un middleware, deberemos pasarle un **callback** a la ruta justo entre el **request** y el **response**.

```
{ router.get('/usuario/:id', callback, usersController.list); }
```

¿Qué es ese callback?

Es nuestro middleware. Aquella funcionalidad que queremos implementar antes de que se ejecute el response de la petición.

CÓMO CONFIGURARLO

Cada middleware que definamos deberá recibir el **request**, el **response** y el **next**.

Para mantener un orden estructural en nuestra aplicación, se sugiere crear el middleware en un archivo aparte y requerirlo en donde se vaya a usar. De esta forma, nos evitamos definir los middlewares como funciones anónimas.

```
function logged(req,res,next) {  
    // funcionalidad  
    next();  
}
```

“

EXPRESS VALIDATOR

Cada vez que le solicitemos al usuario que nos **envíe** información, es importante **validar** esos **datos** tanto desde el *front-end* como desde el *back-end*.



express-validator

Es una librería que nos permite validar campos de una manera sencilla y eficiente.

Para empezar a trabajar con él hace falta instalarlo a través de npm.

```
npm install express-validator --save
```

Al momento de requerir el módulo, nos devolverá un objeto con muchas propiedades. De todas ellas nos interesan sólo tres: check, validationResult y body. Usando destructuring podemos requerirlas para implementarlas en nuestro proyecto.

```
const { check, validationResult, body } = require('express-validator');
```

1. VALIDAR DATOS

PRIMEROS PASOS

En la ruta que va a procesar la información que nos llegue del formulario, deberemos configurar un middleware. ¿Por qué? Porque estas validaciones tienen que ejecutarse **después** de que el usuario envió la información, pero **antes** de que se envíe la respuesta.

Como todo middleware, deberemos enviarlo **entre** el request y el response, sólo que en este caso lo enviaremos como un **array**.

```
router.post('/login', [], userController.login);
```

check()

Es una función que nos va a permitir validar campo por campo. La misma recibe como parámetro un string, que será el nombre del campo que queremos validar.

```
router.post('/login', [
  check('email'),
  ...
], userController.login);
```



Recordar que el **nombre del campo** que le pasemos a check() deberá ser el mismo que hayamos configurado para ese campo en el atributo **name** del formulario.

AGREGANDO VALIDACIONES

Esta librería incluye **métodos** que podemos implementar directamente sobre cada uno de los **checks** que hagamos, para validar diferentes cosas. Algunos de los más usados son:

isLength()

Valida la longitud del campo. Recibe un objeto literal con la propiedad min y max, para configurar la cantidad de caracteres mínimos y máximos que debería tener ese dato.

```
{} check('campo').isLength( {min:6} )
```

isEmpty()

Valida que el dato de ese campo tenga un formato de email correcto.

```
{} check('campo').isEmpty()
```

AGREGANDO VALIDACIONES

isInt()

Valida que el dato de ese campo sea un integer. Adicionalmente, puede recibir un objeto literal con la propiedad min y max, para configurar el número mínimo y máximo que puede ser ese dato.

```
{} check('campo').isInt({min:18, max:99})
```

isEmpty()

Valida que el campo no esté vacío.

```
{} check('campo').isEmpty()
```

Podés encontrar todos los métodos que incluye la librería en [este link](#).

7

Las validaciones con **check()** las haremos dentro del **array** que le pasamos como *parámetro* a la **ruta** que esté **procesando** el formulario.



2. CONFIGURAR ERRORES

validationResult()

Es una **función** que nos va a permitir capturar los datos que no hayan pasado las validaciones que hicimos con **check()**. Recibe el objeto **request**, y **retorna** un objeto con los errores del formulario.

En el método del controlador que esté manejando la petición, crearemos la variable **errors** para almacenar en ella lo que devuelva el método **validationResult()**.

```
const userController = {
  login: (req, res) => {
    let errors = validationResult(req);
  }
}
```

ENVIAR LOS ERRORES

El objeto que devuelve **validationResult()** almacena los errores en la propiedad **errors**, que es un **array**. De modo que, si hubo errores, eso es lo que queremos enviar a la vista.

```
const userController = {
  login: (req, res) => {
    let errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.render('login', {errors: errors.errors});
    }
  }
}
```

11

8

MOSTRAR LOS ERRORES

En la vista que retornamos, tendremos que **recorrer** ese array de errores que nos llegó. Cada error será un objeto con propiedades, en donde, en este escenario, nos interesa la propiedad `msg`, que almacena el mensaje de error que queremos mostrarle al usuario.

```
ejs
<ul>
  <% for(var i = 0; i < errors.length; i++) { %>
    <li> <%= errors[i].msg %> </li>
  <% } %>
</ul>
```

Cuando el usuario ingresa por **primera vez** al formulario, la variable `errores` que enviamos a la vista **no existe**, ya que la petición aún no pasó por el **controlador** que procesa esos datos.



13

MOSTRAR LOS ERRORES

Para evitar fallas, por fuera del for que recorre el array, tendremos que crear un condicional que verifique que la variable `errors` no esté indefinida antes de iniciar el bucle.

```
ejs
<% if(typeof errors != 'undefined') { %>
  <ul>
    <% for(var i = 0; i < errors.length; i++) { %>
      <li> <%= errors[i].msg %> </li>
    <% } %>
  </ul>
<% } %>
```

3. EDITAR MENSAJES

15

withMessage()

Es una función que nos va a permitir configurar el mensaje de error del campo a validar. Recibe un **string**, que será el que le mostraremos al usuario en la vista en caso de que exista un error en ese campo.

```
router.post('/login', [
  check('email').withMessage('El formato ingresado no es válido'),
  ...
], userController.login);
```

Si bien **express-validator** ya trae resueltas muchas validaciones, puede haber ocasiones en las que necesitemos validar datos de una forma más **personalizada**.



17

4.

VALIDACIONES PROPIAS

Al igual que `check()`, las validaciones con **body()** las haremos dentro del **array** que le pasamos como *parámetro* a la **ruta** que esté **procesando** el formulario.



CREANDO VALIDACIONES

Para validar un campo de manera personalizada, haremos uso de dos funciones que nos da la librería:

`body()`

Recibe un string como parámetro, el nombre del campo que queremos validar.

{}

`body('email')`

`custom()`

Recibe un callback con un **valor**. Dentro desarrollaremos la lógica necesaria para validar ese valor. La ejecutamos sobre el campo que queremos validar.

{}

```
body('email').custom(function(value){  
    // lógica a implementar  
})
```

SESSION

Es una **variable** que está accesible en todo el sitio. Nos permite *guardar y compartir* información de un mismo **usuario** entre las vistas.



EXPLICACIÓN TEÓRICA

Por defecto, las solicitudes Express son secuenciales y ninguna solicitud puede vincularse entre sí. No hay forma de saber si esta solicitud proviene de un cliente que ya realizó una solicitud anteriormente.

Los usuarios **no pueden identificarse** a menos que utilicen algún tipo de mecanismo que lo haga posible. Para eso usamos `session`.

Cuando la implementamos, a cada usuario se le asignará una **sesión única** pudiendo de esta manera almacenar el estado de ese usuario.

SESSION EN EXPRESS

Cuando trabajamos con `session` almacenamos, del lado del **servidor**, datos del usuario que sean relevante para permitirle navegar con fluidez por nuestro sitio, desde información personal que sirva para el logueo o alguna característica más global, como el idioma, moneda, o color de fondo.

A su vez, del lado del **cliente**, se generará un identificador único que asociará a ese usuario con toda esa información.

Algo importante a tener en cuenta de `session` es que, cuando el usuario **cierra el navegador**, toda esa información se **borra**. Es decir que los datos de una `session` sólo viven mientras esté abierto el navegador.

3

La **información del usuario** la guardaremos del lado del **servidor**.



El **identificador único** que asocia la información con ese usuario la guardaremos del lado del **cliente**, en el **navegador**.

IMPLEMENTAR SESSION

→ Instalar el módulo `express-session` con npm:

```
>_ npm i express-session --save
```

→ Requerirlo en el entry point de la aplicación:

```
{} const session = require('express-session');
```

→ Configurarlo como middleware a nivel aplicación. Ejecutamos `session()` pasándole como argumento un objeto literal con la propiedad `secret` con un texto único aleatorio, que servirá para identificar nuestro sitio web.

```
{} app.use(session({secret: "Nuestro mensaje secreto"}));
```

4

IMPLEMENTAR SESSION

- Al momento de querer **definir** y **almacenar** información, llamar a la propiedad session del objeto request:

```
{} req.session.colorFondo = 'Violeta';
```

- Para **leer** información de session,

```
{} let colorFondo = req.session.colorFondo;
```

Toda la información que almacenemos en la variable `session` estará disponible para usar en **todas** las vistas del sitio.

“

Son **archivos** que podemos guardar del lado del *cliente*, es decir, en el **navegador** del usuario.



COOKIES

COOKIES EN EXPRESS

A diferencia de la sesión, a las **cookies** les podemos configurar un "tiempo de vida". Es decir que una cookie dejará de existir cuando **expire** ese tiempo y no cuando el usuario cierra el navegador.

Algo a tener en cuenta cuando trabajamos con cookies es que, al estar almacenando datos del lado del cliente, contamos con un límite de espacio.

Es importante destacar que no debemos almacenar ningún dato sensible en una cookie.

Para mantener **logueado** a un usuario luego de cerrar el *navegador*, podemos usar una **cookie** para identificarlo y loguearlo automáticamente la próxima vez que ingrese al sitio.



3

IMPLEMENTAR COOKIES

- Instalar el módulo `cookie-parser` con npm. (Con express-generator ya viene incluido este módulo)

```
>_ npm i cookie-parser --save
```

- Para **crear** una cookie y **guardar** en ella información, ejecutamos el método `cookie()` sobre el objeto `response`, pasándole dos argumentos:
 - ◆ El **nombre** que le quiero asignar a esa cookie
 - ◆ El **valor**

```
{ res.cookie('club', 'Boquita Juniors');
```

El nombre de la cookie que definimos será una propiedad de `cookies`.

IMPLEMENTAR COOKIES

- Para **leer** información de una cookie usamos el objeto `request`, llamando al objeto `cookies`, seguido del nombre de la cookie que definimos anteriormente:

```
{ console.log(req.cookies.club);
```

5

6

Módulo: MySQL TABLAS

1.

ESTRUCTURA

ATRIBUTOS

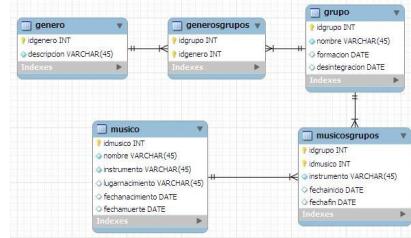
Son las **características** que van a definir a cada entidad. Por ejemplo, la entidad **Películas** podría tener los siguientes atributos:

PELÍCULAS

título
rating
fecha_estreno
país

DIAGRAMA RELACIONAL

Estos diagramas permiten entender de forma rápida y sencilla **todas** las tablas de nuestra base de datos.



ENTIDAD

Dentro de nuestro sistema tendremos **entidades**, que serán la representación de un objeto o cosa de la vida real. Para integrarlos en nuestro diagrama, los representaremos usando un rectángulo. Para el nombre de estas tablas usaremos sustantivos en **plural**.

PELÍCULAS

ACTORES

GÉNEROS

CLAVE PRIMARIA

Una **Clave Primaria** o **Primary Key** es un campo (o una combinación de campos) que identifica a cada fila de una tabla de **forma única**. Es decir que no puede haber dos filas en una tabla que tengan la misma PK.

No es obligatorio tener una, pero es altamente recomendable que cada fila dentro de una tabla tenga cierta unicidad.

id	título	rating	fecha_estreno	país
1001	Pulp Fiction	9.8	1995-02-16	Estados Unidos
1002	Kill Bill Vol. 1	9.5	2003-11-27	Estados Unidos

CLAVE PRIMARIA

Para identificar la clave primaria en una entidad, podemos escribir el atributo en negrita seguido de las iniciales **PK** entre paréntesis.

PELÍCULAS

id (**PK**)
título
rating
fecha_estreno
país

Módulo: MySQL
RELACIONES

QUÉ SON

Las relaciones indican cómo se van a relacionar dos tablas. Dentro de una base de datos existen 3 tipos de relaciones:

- de uno a uno
- de uno a muchos
- de muchos a muchos

¿Cómo podemos saber cómo se relaciona una entidad con otra?

Planteando un ejemplo concreto que nos ayude a definir cómo interactúan esas dos entidades entre sí.

2

CARDINALIDAD

Es la forma en que se relacionan las entidades.

Cardinalidad	Se lee	Representación
1:1	Uno a uno	+
1:M	Uno a muchos	+
M:1	Muchos a uno	≥
M:M	Muchos a muchos	≥≥

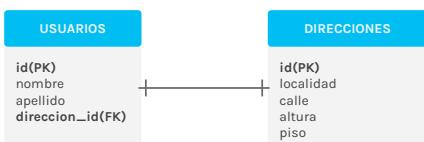
3

TIPOS DE RELACIONES

UNO A UNO 1:1

Un usuario tiene sólo una dirección. Una dirección pertenece sólo a un usuario.

Para establecer la relación colocamos la clave primaria de la dirección en la tabla de usuarios, indicando que esa dirección está asociada a ese usuario.

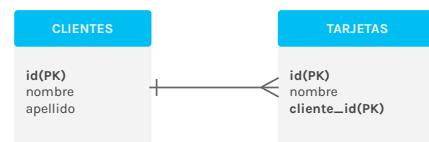


5

UNO A MUCHOS 1:M

Un cliente puede tener muchas tarjetas. Una tarjeta pertenece sólo a un cliente.

Para establecer la relación colocamos la clave primaria del cliente en la tabla de tarjetas, indicando que esa tarjeta está asociada a un usuario en particular.



6

MUCHOS A MUCHOS M:M

Un **cliente** puede comprar **muchos productos**. Un **producto** puede ser comprado por **muchos clientes**.

En las relaciones **M:M**, en la base de datos, la relación en sí pasa a ser una **tabla**. Esta tabla intermedia -también conocida como tabla pivot- tiene como mínimo 3 datos: una clave primaria (**PK**) y dos claves foráneas (**FK**), cada una haciendo referencia a cada tabla de la relación.



7

MUCHOS A MUCHOS M:M

En este ejemplo **cliente_producto** sería nuestra tabla intermedia. Cada fila de esta tabla representa un cruce entre cliente y producto -podría ser en este caso, una compra:

- La fila 1 indica que el **cliente 1 (Juan)** compró el **producto 1 (Pelota)**
- La fila 2 indica que **Juan** también compró el **producto 2 (Laptop)**
- La fila 3 indica que una **Laptop** también fue comprada por el cliente 3 (**Marta**)

CLIENTES			CLIENTE_PRODUCTO			PRODUCTOS	
id	nombre	apellido	id	producto_id	cliente_id	id	nombre
1	Juan	Perez	1	1	1	1	Pelota
2	Clara	Sanchez	2	2	1	2	Laptop
3	Marta	Garcia	3	2	3	3	Celular

8

Módulo: MySQL

TIPOS DE DATOS

1.

DATOS DE TIPO NUMÉRICO

NUMÉRICOS CON DECIMALES

FLOAT

Permite almacenar pequeños números decimales

DOUBLE

Permite almacenar grandes números decimales

DECIMAL

Permite almacenar grandes números decimales de punto fijo

TIPOS DE DATOS

Los datos o atributos de cada registro de una tabla tienen que ser de un tipo de dato concreto.

Cuando diseñamos una base de datos tenemos que pensar qué tipo de dato nos sirve para nuestro modelo.

Cada tipo de dato tiene un tamaño determinado y cuanta más precisión pongamos a este tipo de dato, más rápido y performante va a funcionar MySQL.

012345

Tipos numéricos

2020-05-09

Tipos de fecha

Ad lorem ipsum

Tipos de texto

NUMÉRICOS SIN DECIMALES

TINYINT

-128 a 128, 0 a 255

SMALLINT

-32768 a 32767, 0 a 65535

MEDIUMINT

-8388608 a 8388607, 0 a 16777215

INT

-2147483648 to 214748364, 0 a 4294967295

BIGINT

-9223372036854775808 a 9223372036854775807

0 a 18446744073709551615

BOOLEANS

MySQL guarda los booleanos por detrás como un **cero** o como un **uno**. Por cuestiones de performance, no se recomienda utilizar este tipo de dato en MySQL.

En caso de querer guardar valores "verdaderos" y "falsos" podemos usar el tipo de dato **tinyint** y usar el 0 para representar el **false** y el 1 para representar el **true**.

2.

DATOS DE TIPO FECHA

DATOS DE TIPO FECHA

A la hora de almacenar fechas, hay que tener en cuenta que MySql no comprueba de una manera estricta si una fecha es válida o no.

DATE

Almacena solamente la fecha en formato YYYY-MM-DD

TIME

Almacena solamente la hora en formato HH:MM:SS

DATETIME

Corresponde a una representación completa de fecha y hora, es decir algo como el 03 de enero de 1967 a las 8:00 am, lo que se almacena del siguiente modo **1967-01-03 08:00:00**

8

3.

DATOS DE TIPO TEXTO

DATOS DE TIPO TEXTO

CHAR(num)

El número entre paréntesis va a indicar la **cantidad exacta** de caracteres.

Ejemplo: char(100) → 100 caracteres **siempre**

VARCHAR(num)

El número entre paréntesis va a indicar la **cantidad máxima** de caracteres.

Ejemplo: varchar(100) → 100 caracteres **como máximo**

TEXT

Determina un dato de tipo texto sin límite de caracteres. Se suele usar para post de un blog, por ejemplo.

10

4.

CONSTRAINTS

Son **restricciones** a nivel tabla. Se especifican vía DDL.
El servidor las analiza a la hora de modificar registros.

CONSTRAINTS

UNIQUE KEYS

Una unique key es una restricción que solo permite valores únicos para uno (o múltiples) campos.

NULL / NOTNULL

NULL Significa que el valor para ese campo no existe o no se conoce. NULL NO es vacío '' (blank) ni cero (0).

DEFAULT

Se usa para definir un **valor** por defecto para una columna. Este valor se le va a agregar a cada registro nuevo siempre y cuando no se especifique otro valor que lo sobreescriba.

12

CONSTRAINTS

AUTO_INCREMENT

Genera un número único y lo incrementa automáticamente con cada nuevo registro en la tabla. Se suele usar a menudo para el campo **ID**.

Módulo: MySQL

CREATE, DROP Y ALTER

CREATE TABLE

Con **CREATE TABLE** podemos crear una tabla desde cero, junto con sus columnas, sus tipos y sus constraints.

```
SQL CREATE TABLE nombre_de_la_tabla (
    nombre_de_columna_1 TIPO_DE_DATO CONSTRAINT,
    nombre_de_columna_2 TIPO_DE_DATO CONSTRAINT,
);
```

```
SQL CREATE TABLE post (
    id INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
    titulo VARCHAR(200),
);
```

CREATE TABLE EJEMPLO

```
SQL CREATE TABLE movies (
    id INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(500) NOT NULL,
    rating DECIMAL(3,1) UNSIGNED NOT NULL,
    awards INT UNSIGNED DEFAULT 0,
    release_date DATE NOT NULL,
    length INT UNSIGNED NOT NULL
);
```

FOREIGN KEY

Cuando creamos una columna que contenga una id foránea, será necesario usar la sentencia **FOREIGN KEY** para aclarar a qué tabla y a qué columna hace referencia aquel dato.

*Es importante remarcar que la tabla **clientes** deberá existir antes de correr esta sentencia para crear la tabla **ordenes**.*

```
SQL CREATE TABLE ordenes (
    orden_id INT NOT NULL,
    orden_numero INT NOT NULL,
    cliente_id INT,
    PRIMARY KEY (orden_id),
    FOREIGN KEY (cliente_id) REFERENCES clientes(id)
);
```

DROP TABLE

DROP TABLE borrará la tabla que le especifiquemos en la sentencia.

```
SQL DROP TABLE IF EXIST movies;
```

ALTER TABLE

ALTER TABLE permite alterar una tabla ya existente y va a operar con tres comandos:

- **ADD**, para agregar una columna
- **MODIFY**, para modificar una columna
- **DROP**, para borrar una columna

```
ALTER TABLE nombre_de_tabla
ADD columna3 TIPO_DE_DATO [FIRST|AFTER columna2]
MODIFY NUEVO_TIPO_DE_DATO
DROP columna4;
```

ALTER TABLE

SQL

```
ALTER TABLE movies  
ADD rating DECIMAL(3,1) UNSIGNED NOT NULL;
```

Agrega la columna `rating`, aclarando tipo de dato y constraint.

SQL

```
ALTER TABLE movies  
MODIFY rating DECIMAL(4,1) UNSIGNED NOT NULL;
```

Modifica el decimal de la columna `rating`. Aunque el resto de las configuraciones de la tabla no se modifiquen, es necesario escribirlas en la sentencia.

SQL

```
ALTER TABLE movies  
DROP rating;
```

Borra la columna `rating`.

Módulo: MySQL
**INSERT, UPDATE,
DELETE**

INSERT

Existen dos formas de agregar datos en una tabla:

- Insertando datos en **todas las columnas**
- Insertando datos en las **columnas que especificuemos**

TODAS LAS COLUMNAS

Si estamos insertando datos en todas las columnas, no hace falta aclarar los nombres de cada columna. Sin embargo, el orden en el que insertemos los valores, deberá ser el mismo orden que tengan asignadas las columnas en la tabla.

SQL

```
INSERT INTO table_name (columna_1, columna_2, columna_3, ...)
VALUES (valor_1, valor_2, valor_3, ...);
```

SQL

```
INSERT INTO usuarios (id, nombre, apellido,
VALUES (DEFAULT, 'Diego', 'Díaz');
```

SQL

```
INSERT INTO usuarios
VALUES (DEFAULT, 'Diego', 'Díaz');
```

COLUMNAS ESPECÍFICAS

Para insertar datos en una columna en específico, aclaramos la tabla y luego escribimos el nombre de la o las columnas entre los paréntesis.

SQL

```
INSERT INTO usuarios (nombre)
VALUES ('Santi');
```

SQL

```
INSERT INTO peliculas (duracion, titulo)
VALUES (112, 'Kill Bill');
```

3

5

DELETE

Con **DELETE** podemos borrar información de una tabla. Es importante recordar utilizar siempre el **WHERE** en la sentencia para agregar la condición de cuáles son las filas que queremos eliminar. Si no escribimos el **WHERE** estaríamos borrando **toda la tabla** y no un registro en particular.

SQL

```
DELETE FROM nombre_tabla WHERE condicion;
```

SQL

```
DELETE FROM usuarios WHERE usuario_id = 4;
```

UPDATE

UPDATE modificará los registros existentes de una tabla. Al igual que con **DELETE**, es importante no olvidar el **WHERE** cuando escribimos la sentencia, aclarando la condición.

SQL

```
UPDATE nombre_tabla
SET columna_1 = valor_1, columna_2 = valor_2, ...
WHERE condicion;
```

SQL

```
UPDATE usuarios
SET nombre = 'Alfredo', apellido = 'Caseros'
WHERE id = 1;
```

2

4

6

Módulo: MySQL
SELECT

CÓMO USARLO

Toda consulta a la base de datos va a empezar con la palabra **SELECT**. Su funcionalidad es la de realizar consultas sobre **una o varias columnas** de una tabla.

Para especificar sobre qué tabla queremos realizar esa consulta usamos la palabra **FROM** seguida del nombre de la tabla.

SQL

```
SELECT nombre_columna, nombre_columna, ...
FROM nombre_tabla;
```

EJEMPLO

id	titulo	rating	fecha_estreno	país
1001	Pulp Fiction	9.8	1995-02-16	Estados Unidos
1002	Kill Bill Vol. 1	9.5	2003-11-27	Estados Unidos

Para conocer los títulos y ratings de las películas guardadas en la tabla **películas**, podríamos hacerlo ejecutando la siguiente consulta:

SQL

```
SELECT titulo, rating
FROM peliculas;
```

Módulo: MySQL
WHERE y ORDER BY

1.

WHERE

CÓMO USARLO

La funcionalidad del **WHERE** es la de condicionar y filtrar las consultas **SELECT** que se realizan a una base de datos.

SQL

```
SELECT nombre_columna_1, nombre_columna_2, ...
FROM nombre_tabla
WHERE condicion;
```

Teniendo una tabla **usuarios**, podría consultar nombre y edad, filtrando con un **WHERE** sólo los usuarios **mayores de 17 años** de la siguiente manera:

SQL

```
SELECT nombre, edad
FROM usuarios
WHERE edad > 17;
```

3

OPERADORES

=	Igual a
>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que
<>	Diferente a
!=	Diferente a

4

OPERADORES

IS NULL	→ Es nulo
BETWEEN	→ Entre dos valores
IN	→ Lista de valores
LIKE	→ Se ajusta a...

QUERY'S DE EJEMPLO

SQL

```
SELECT *
FROM movies
WHERE release_date > '2000-01-01';
```

SQL

```
SELECT *
FROM movies
WHERE title LIKE 'Avatar';
```

5

6

QUERYS DE EJEMPLO

```
SQL
SELECT *
FROM movies
WHERE awards >= 3
AND awards < 8;
```

```
SQL
SELECT *
FROM movies
WHERE awards = 2
OR awards = 6;
```

QUERYS DE EJEMPLO

```
SQL
DELETE FROM usuarios
WHERE id = 2;
```

 Si en esta query quitáramos el WHERE...
¡BORRARÍAMOS TODA LA TABLA!

7

8

2. ORDER BY

CÓMO USARLO

ORDER BY se utiliza para ordenar los resultados de una consulta **según el valor de la columna especificada**. Por defecto, se ordena de forma ascendente (ASC) según los valores de la columna. También se puede ordenar de manera descendente (DESC) aclarándolo en la consulta.

```
SQL
SELECT nombre_columna1, nombre_columna2
FROM tabla
WHERE condicion
ORDER BY nombre_columna1;
```

9

10

QUERY DE EJEMPLO

Teniendo una tabla **usuarios**, podría consultar los nombres, filtrar con un WHERE sólo los usuarios **mayores de 21 años** y ordenarlos de forma descendente tomando como referencia la columna nombre.

```
SQL
SELECT nombre, edad
FROM usuarios
WHERE edad > 21
ORDER BY nombre DESC;
```

11

Módulo: MySQL
LIMIT Y OFFSET

1. **LIMIT**

CÓMO USARLO

Su funcionalidad es la de **limitar el número de filas** (registros/resultados) devueltas en las consultas SELECT. También establece el **número máximo** de registros a eliminar con DELETE.

SQL

```
SELECT nombre_columna1, nombre_columna2
FROM nombre_tabla
LIMIT cantidad_de_registros;
```

QUERY DE EJEMPLO

Teniendo una tabla **películas** podríamos armar un top 10 con las películas que tengan más de 4 premios usando un **LIMIT** en la siguiente consulta:

SQL

```
SELECT *
FROM películas
WHERE premios > 4
LIMIT 10;
```

3

4

2. **OFFSET**

CÓMO USARLO

En un escenario en donde hacemos una consulta de todas las películas de la base de datos, la misma nos devolvería muchos registros. Usando un **LIMIT** podríamos aclarar un límite de 20. ¿Pero cómo haríamos si quisieramos recuperar sólo 20 películas pero saltando las primeras 10 de la tabla? **OFFSET** nos permite especificar a partir de qué fila comenzar la recuperación de los datos solicitados.

6

Consulta sql

```
SELECT id, nombre, apellido  
FROM alumnos  
LIMIT 20  
OFFSET 20;
```

Consulta sql

```
SELECT id, nombre, apellido  
FROM alumnos  
LIMIT 20  
OFFSET 20;
```

• Seleccionamos las columnas id, nombre y apellido...

7

8

Consulta sql

```
SELECT id, nombre, apellido  
FROM alumnos  
LIMIT 20  
OFFSET 20;
```

Consulta sql

```
SELECT id, nombre, apellido  
FROM alumnos  
LIMIT 20  
OFFSET 20;
```

• Limitamos los registros de la tabla resultante a 20 registros.

de la tabla alumnos.

9

10

Consulta sql

```
SELECT id, nombre, apellido  
FROM alumnos  
LIMIT 20  
OFFSET 20;
```

Desplazamos los resultados 20 posiciones para que se muestre desde la posición 21.

11

Módulo: MySQL
BETWEEN y LIKE

1. **BETWEEN**

CUÁNDO USARLO

Cuando necesitamos obtener valores **dentro de un rango**, usamos el operador BETWEEN.

- BETWEEN incluye los extremos.
- BETWEEN funciona con **números, textos y fechas**.
- Se usa como un filtro de un WHERE.

Por ejemplo, coloquialmente:

- Dados los números: 4, 7, 2, 9, 1
- Si hiciéramos un BETWEEN entre 2 y 7 devolvería 4, 7, 2 (excluye el 9 y el 1, e incluye el 2)

QUERY DE EJEMPLO

Con la siguiente consulta estaríamos seleccionando **nombre** y **edad** de la tabla **alumnos** sólo cuando las edades estén **entre** 6 y 12.

```
SQL
SELECT nombre, edad
FROM alumnos
WHERE edad BETWEEN 6 AND 12;
```

3

4

2. **LIKE**

CÓMO USARLO

Cuando hacemos un filtro con un WHERE, podemos especificar un patrón de búsqueda que nos permita especificar algo concreto que queremos encontrar en los registros. Eso lo logramos utilizando **comodines** (wildcards).

Por ejemplo, podríamos querer buscar:

- Los nombres que tengan la letra 'a' como segundo carácter
- Las direcciones postales que incluyan la calle 'Monroe'
- Los clientes que empiecen con 'Los' y terminen con 's'

6

COMODÍN %

Es un sustituto que representa **cero, uno, o varios** caracteres.



COMODÍN _

Es un sustituto para **un sólo** carácter.



QUERY'S DE EJEMPLO

SQL

```
SELECT nombre  
FROM usuarios  
WHERE edad LIKE '_a%';
```

Devuelve aquellos nombres que tengan la letra 'a' como segundo caracter.

SQL

```
SELECT nombre  
FROM usuarios  
WHERE direccion LIKE '%Monroe%';
```

Devuelve las direcciones de los usuarios que incluyan la calle 'Monroe'.

QUERY'S DE EJEMPLO

SQL

```
SELECT nombre  
FROM clientes  
WHERE nombre LIKE 'Los%s';
```

Devuelve los clientes que empiecen con 'Los' y terminen con 's'.

Módulo: MySQL
ALIAS

CÓMO USARLO

Los **ALIAS** se usan para darle un nombre temporal y más amigable a las **tablas**, **columnas** y **funciones**. Los **alias** se definen durante una consulta y persisten **sólo** durante esa consulta.

Para definir un alias usamos las iniciales **AS** precediendo a la columna que estamos queriendo asignarle ese alias.

SQL

```
SELECT nombre_columna1 AS alias_nombre_columna1
FROM nombre_tabla;
```

Alias para una **COLUMNA**



Consulta sql

```
SELECT razon_social_cliente AS nombre
FROM cliente
WHERE nombre LIKE 'a%';
```

Consulta sql

```
SELECT razon_social_cliente AS nombre
FROM cliente
WHERE nombre LIKE 'a%';
```

Seleccionamos la columna
razon_social_cliente y le
asignamos el **ALIAS** nombre.

Consulta sql

```
SELECT razon_social_cliente AS nombre
FROM cliente
WHERE nombre LIKE 'a%';
```

En el **FROM** elegimos tabla
cliente.
Con el **WHERE** filtramos los
registros de la columna nombre
que empiecen con la letra a.

Alias para una TABLA



Consulta sql

```
SELECT nombre, apellido, edad  
FROM alumnos_comision_inicial AS alumnos;
```

8

Consulta sql

```
SELECT nombre, apellido, edad  
FROM alumnos_comision_inicial AS alumnos;
```

Selecciónamos las columnas nombre, apellido y edad.

Consulta sql

```
SELECT nombre, apellido, edad  
FROM alumnos_comision_inicial AS alumnos;
```

Hacemos la consulta sobre la tabla `alumnos_comision_inicial` y le asignamos el ALIAS `alumnos`.

Para asignar un alias con espacio es necesario escribirlo entre comillas simples:
`FROM alumnos_comision_inicial AS 'Los alumnos';`

9

10

De este modo, podemos darle alias a las **columnas** y **tablas** que vamos trayendo y hacer más legible la manipulación de datos, teniendo siempre presente que los alias **no modifican** los nombres originales en la base de datos.



Módulo: MySQL
TABLE REFERENCE

TABLE REFERENCE

Hasta ahora vimos consultas (SELECT) dentro de una **tabla**. Pero también es posible y necesario hacer consultas a distintas tablas y unir los resultados.

Por ejemplo, un posible escenario sería querer consultar una tabla en donde están los **datos de los clientes** y otra tabla en donde están los **datos de las ventas a esos clientes**.



TABLE REFERENCE

Seguramente, en la tabla de **ventas**, existirá un campo con el id del cliente (**cliente_id**).

Si quisiera mostrar **todas** las ventas de un cliente concreto, necesitaré usar datos de **ambas tablas** y **vincularlas** con algún **campo que comparten**. En este caso, el **cliente_id**.

CLIENTES	VENTAS
id (PK) nombre apellido	id (PK) fecha local cliente_id (FK)

Consulta sql

```

SELECT clientes.id AS id, clientes.nombre, ventas.fecha
FROM clientes, ventas
WHERE clientes.id = ventas.cliente_id;
  
```

3

2

Consulta sql

```

SELECT clientes.id AS id, clientes.nombre, ventas.fecha
FROM clientes, ventas
WHERE clientes.id = ventas.cliente_id;
  
```

Seleccionamos:

- La columna **id** de la tabla **clientes** y le asigno el alias **id**.
- La columna **nombre** de la tabla **clientes**.
- La columna **fecha** de la tabla **ventas**.

Consulta sql

```

SELECT clientes.id AS id, clientes.nombre, ventas.fecha
FROM clientes, ventas
WHERE clientes.id = ventas.cliente_id;
  
```

5

4

El select lo hacemos sobre las tablas **clientes** y **ventas**.

Hasta acá la consulta traerá **todos los clientes y todas las ventas**. Por eso nos falta todavía agregar un **filtro** que muestre **solo las ventas de cada usuario** en particular.

Consulta sql

```
SELECT clientes.id AS id, clientes.nombre, ventas.fecha  
FROM clientes, ventas  
WHERE clientes.id = ventas.cliente_id;
```

En el **WHERE** creamos una condición para traer aquellos registros en donde el **id** del cliente sea igual en **ambas tablas**.

7

TABLE REFERENCE

```
SELECT Clientes.id AS id, Clientes.nombre,  
Ventas.fecha  
FROM Clientes, Ventas  
WHERE Clientes.id = Ventas.ClienteID
```

Veamos paso a paso qué hace esta consulta...

8

TABLE REFERENCE

```
SELECT Clientes.id AS id, Clientes.nombre,  
Ventas.fecha
```

Selecciono la columna **id** de la tabla **clientes**, y le pongo un alias **id**, la columna nombre de la tabla **Clientes**, y la columna fecha de la tabla **Ventas**.

9

TABLE REFERENCE

```
FROM Clientes, Ventas
```

De las tablas Clientes y Ventas...

Hasta acá la consulta traería **todos** los clientes asociados a **todas** las ventas. Por eso nos falta todavía agregar un filtro que muestre las **ventas de cada usuario**

10

TABLE REFERENCE

```
WHERE Clientes.id = Ventas.ClienteID
```

Filtro los registros tal que el ID del cliente sea igual en ambas tablas. Igualo la columna id (Clientes.id) de la tabla Clientes y la columna ClienteID de la tabla Ventas.

11

TABLE REFERENCE

clientes
id
nombre
apellido
edad
ventas_id

ventas
id
fecha
local

12

TABLE REFERENCE

```
SELECT Clientes.id AS id, Clientes.nombre,  
Ventas.fecha  
FROM Clientes, Ventas  
WHERE Clientes.id = Ventas.ClienteID
```

Ahora se entiende! :D

Módulo: MySQL
JOINS

PORQUÉ USAR JOINS

Además de hacer consultas dentro de una tabla o hacia muchas tablas a través de **table reference**, también es posible y necesario hacer consultas a **distintas tablas**, y unir esos resultados con **JOINS**.

Si bien cumplen la misma función que **table reference**, los **JOINS**:

- Proveen ciertas flexibilidades adicionales
- Su sintaxis es mucho más utilizada
- Presentan una mejor performance.

INNER JOIN

El **INNER JOIN** hará una **cruz**a entre dos tablas. Si cruzáramos las tablas de **clientes** y **ventas** y hubiese algún cliente **sin ventas**, el **INNER JOIN** **no traería** a ese cliente como resultado.

INNER JOIN

CLIENTES		
id	nombre	apellido
1	Juan	Perez
2	Clara	Sanchez
3	Marta	García

VENTAS		
id	cliente_id	fecha
1	2	12/03/2019
2	2	22/08/2019
3	1	04/09/2019

3

LEFT JOIN - RIGHT JOIN

Estos tipos de **JOINS** **no excluyen** resultados de alguna de las dos tablas. Si hubiese clientes **sin ventas** podríamos incluirlos en el resultado mediante **LEFT** o **RIGHT JOIN**.



CREANDO UN INNER JOIN

Antes escribíamos:

```
SQL      SELECT clientes.id AS id, clientes.nombre, ventas.fecha
              FROM clientes, ventas
```

Ahora escribiremos:

```
SQL      SELECT clientes.id AS id, clientes.nombre, ventas.fecha
              FROM clientes
              INNER JOIN ventas
```

Si bien ya dimos el primer paso que **es cruzar** ambas tablas, aún nos falta aclarar **dónde** está ese cruce.

Es decir, qué **clave primaria (PK)** se cruzará con qué **clave foránea (FK)**.



CREANDO UN INNER JOIN

La sintaxis del join **no utiliza** el **WHERE**, si no que **requiere** la palabra **ON**. Es ahí en donde indicaremos el **filtro** a tener en cuenta para realizar el cruce.

Es decir que lo que antes escribímos en el **WHERE** ahora lo escribiremos en el **ON**.

```
SQL
SELECT clientes.id AS id, clientes.nombre, ventas.fecha
FROM clientes
INNER JOIN ventas
ON clientes.id = ventas.cliente_id
```

¿Y si quisiéramos **incluir** en el resultado aquellos **clientes** que **NO** tengan **ventas** asociadas?



7

CREANDO UN LEFT JOIN

Para incluir aquellos clientes sin ventas basta cambiar **INNER JOIN** por **LEFT JOIN**. El **LEFT JOIN** incluirá **todos** los registros de la primera tabla de la consulta (la tabla **izquierda**) incluso cuando no exista coincidencia con la tabla derecha.

```
SQL
SELECT clientes.id AS id, clientes.nombre, ventas.fecha
FROM clientes
LEFT JOIN ventas
ON clientes.id = ventas.cliente_id
```



¿Y para **incluir** en el resultado aquellas **ventas** que **NO** tienen **clientes** asociados?



9

CREANDO UN RIGHT JOIN

Para incluir aquellas ventas sin clientes basta cambiar **LEFT JOIN** por **RIGHT JOIN**. El **RIGHT JOIN** incluirá **todos** los registros de la tabla **derecha**. Si miramos la query, la tabla **ventas** aparece posterior a la tabla de **clientes...a la derecha!**

```
SQL
SELECT clientes.id AS id, clientes.nombre, ventas.fecha
FROM clientes
RIGHT JOIN ventas
ON clientes.id = ventas.cliente_id
```



CRUZANDO MUCHAS TABLAS

En el siguiente ejemplo se ve cómo hacer cruces de muchas tablas en una misma consulta usando **joins**:

```
SELECT clientes.id AS id, clientes.nombre, ventas.fecha
FROM clientes
INNER JOIN ventas
ON clientes.id = ventas.cliente_id
INNER JOIN productos
ON productos.id = ventas.producto_id
```

SQL

11

12

Módulo: MySQL

DISTINCT

CÓMO FUNCIONA

La cláusula **DISTINCT** nos devuelve **valores únicos**. En una tabla, una columna puede contener valores duplicados y algunas veces sólo se necesita un listado con los valores diferentes, es decir, que no aparezcan aquellos que están repetidos.

SQL

```
SELECT DISTINCT columna_1, columna_2  
FROM nombre_tabla;
```

DISTINCT - EJEMPLO

Partiendo de una tabla de **usuarios**, si ejecutáramos la consulta:

```
SQL SELECT pais FROM usuarios;
```

Obtendríamos:

'Perú, Perú, Argentina, Francia, Argentina'

Existen escenarios en los que vamos a necesitar obtener sólo los valores **distintos** que aparecen en una columna. Agregando el **DISTINCT** en la consulta:

```
SQL SELECT DISTINCT pais FROM usuarios;
```

Obtendríamos:

'Perú, Argentina, Francia'

Consulta sql

```
SELECT DISTINCT actors.first_name, actors.last_name  
FROM actors  
INNER JOIN actor_movie ON actors.id = actor.movie.actor_id  
INNER JOIN movies ON movies.id = actor_movie.movie_id  
WHERE movies.title LIKE '%Harry Potter%';
```

En este ejemplo vemos una query que pide los actores que hayan actuado en **cualquier película de Harry Potter**. Si no escribiéramos el **DISTINCT** los actores que hayan participado en más de una película, aparecerían repetidos en el resultado.

Módulo: MySQL
FUNCIONES MySQL

CONCAT

Usamos **CONCAT** para **concatenar** dos o más expresiones:

SQL `SELECT CONCAT('Hola ', 'a ', 'todos.');`
 > 'Hola a todos.'

SQL `SELECT CONCAT('La respuesta es: ', 24, '.');`
 > 'La respuesta es 24.'

SQL `SELECT CONCAT('Nombre: ', first_name, ' ', last_name)
FROM actors;`
 > 'Nombre: Emilia Clarke'

3

COALESCE

Usamos **COALESCE** para obtener la **primera expresión** que no sea **NULL**:

SQL `SELECT COALESCE(NULL, 1, 20, 'Digital House');`
 > 1

SQL `SELECT COALESCE(NULL, NULL, 'Digital House');`
 > 'Digital House'

COALESCE

Los tres clientes de la siguiente tabla poseen uno o más datos nulos:

CLIENTES				
id	nombre	celular	casa	trabajo
1	Juan	124	345	980
2	Rocío		187	243
3	Matías			428

4

COALESCE

Usando **COALESCE** podremos obtener el **primer dato no nulo** de cada registro, aclarando las columnas a tener en cuenta.

SQL `SELECT id, nombre, COALESCE(celular, casa, trabajo) AS telefono
FROM clientes;`

id	nombre	telefono
1	Juan	124
2	Rocío	187
3	Matías	428

5

DATEDIFF

Usamos **DATEDIFF** para devolver la **diferencia** entre dos fechas, tomando como granularidad el intervalo especificado.

SQL `SELECT DATEDIFF(hour, '2017/08/25 07:00', '2017/08/25 12:45');`
 > 5

Devuelve 5 porque es la cantidad de horas de diferencia entre las 7 y las 12:45. Esta información da un resultado aproximado.

SQL `SELECT DATEDIFF(minute, '2017/08/25 07:00', '2017/08/25 12:45');`
 > 345

Devuelve 345 porque es la cantidad de minutos que van desde las 7 hasta las 12:45 (300min + 45min).

6

EXTRACT

Usamos **EXTRACT** para **extraer** partes de una fecha:

```
SQL SELECT EXTRACT(SECOND FROM '2014-02-13 08:44:21');  
       > 21
```

```
SQL SELECT EXTRACT(MINUTE FROM '2014-02-13 08:44:21');  
       > 44
```

```
SQL SELECT EXTRACT(HOUR FROM '2014-02-13 08:44:21');  
       > 8
```

```
SQL SELECT EXTRACT(DAY FROM '2014-02-13 08:44:21');  
       > 13
```

EXTRACT

```
SQL SELECT EXTRACT(WEEK FROM '2014-02-13 08:44:21');  
       > 6
```

```
SQL SELECT EXTRACT(MONTH FROM '2014-02-13 08:44:21');  
       > 2
```

```
SQL SELECT EXTRACT(QUARTER FROM '2014-02-13 08:44:21');  
       > 1
```

```
SQL SELECT EXTRACT(YEAR FROM '2014-02-13 08:44:21');  
       > 2014
```

7

8

REPLACE

Usamos **REPLACE** para reemplazar una secuencia de caracteres por otra en un string.

```
SQL SELECT REPLACE('abc abc', 'a', 'B');  
       > Bbc Bbc
```

```
SQL SELECT REPLACE('abc abc', 'A', 'B');  
       > abc abc
```

```
SQL SELECT REPLACE('123 123', '2', '5');  
       > 153 153
```

DATE FORMAT

Usamos **CASE**, para evaluar condiciones, y devolver la primera que se cumpla.

```
SQL SELECT DATE_FORMAT('2017-06-15', '%Y');  
       > '2017'
```

```
SQL SELECT DATE_FORMAT('2017-06-15', '%W %M %e %Y');  
       > 'Thursday June 15 2017'
```

9

10

CASE

Usamos **CASE** para **evaluar condiciones**, y devolver la primera que se cumpla. En este ejemplo, la tabla resultante tendrá 4 columnas: *id*, *title*, *rating*, *rating_categories*. Esta última mostrará 'Mala', 'Regular', etc, **según el rating** de la película.

```
SQL SELECT id, title, rating  
CASE  
    WHEN rating < 4 THEN 'Mala'  
    WHEN rating < 6 THEN 'Regular'  
    WHEN rating < 8 THEN 'Buena'  
    WHEN rating < 9.5 THEN 'Muy buena'  
    ELSE 'Excelente'  
END AS rating_categories  
FROM movies  
ORDER BY rating
```

11

“

Módulo: MySQL

FUNCIONES DE AGREGACIÓN

Las funciones de agregación realizan **cálculos** sobre un conjunto de datos y **devuelven** un **único resultado**. Excepto **COUNT**, las funciones de agregación **ignorarán** los valores **NULL**.



COUNT

Devolverá la cantidad de **filas/registros** que cumplen con el criterio.

SQL

```
SELECT COUNT(*) FROM movies;
```

Devolverá la cantidad de registros de la tabla movies

SQL

```
SELECT COUNT(id) AS total FROM movies WHERE genre_id = 3;
```

Devolverá la cantidad de películas de la tabla movies con el genero_id 3 En una columna nombrada total

AVG, SUM

AVG (average) devolverá el promedio de una columna con valores numéricos.

SUM (suma) devolverá la suma de una columna con valores numéricos.

SQL

```
SELECT AVG(rating) FROM movies;
```

Devolverá el promedio del rating de las películas de la tabla movies.

SQL

```
SELECT SUM(length) FROM movies;
```

Devolverá la suma de las duraciones de las películas de la tabla movies

3

4

MIN, MAX

MIN devolverá el valor mínimo de una columna con valores numéricos.

MAX devolverá el valor máximo de una columna.

SQL

```
SELECT MIN(rating) FROM movies;
```

Devolverá el rating de la película menos ranqueada.

SQL

```
SELECT MAX(rating) FROM movies;
```

Devolverá el rating de la película mejor ranqueada.

5

Módulo: MySQL
GROUP BY

GROUP BY - SINTAXIS

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
```

GROUP BY se usa para **agrupar los registros** de la tabla resultante de una consulta por una o más columnas.

GROUP BY - EJEMPLO

```
SELECT marca
FROM autos
GROUP BY marca
```

3

GROUP BY - EJEMPLO

id	marca	modelo	marca
1	Renault	Clio	Renault
2	Renault	Megane	Seat
3	Seat	Ibiza	Opel
4	Seat	Leon	
5	Opel	Corsa	
6	Renault	Clio	

4

GROUP BY - SINTAXIS

Dado que **GROUP BY** agrupa la información, perdemos el detalle de cada una de las filas. Es decir, ya no nos interesa el valor de cada fila sino un resultado consolidado entre todas las filas.

La consulta:

```
SELECT id, marca
FROM autos
GROUP BY marca
```

Nos daría un error. Si agrupamos los datos por **marca**, ya no podemos pedir el campo **id**

GROUP BY - SINTAXIS

Por ende, al utilizar **GROUP BY**, en los campos que se muestran como resultado en el **SELECT** solamente podemos indicar:

- Datos agrupados
- Funciones de agregación

Veamos algunos ejemplos...

5

2

4

6

GROUP BY - EJEMPLO

```
SELECT marca, MAX(precio)
FROM autos
GROUP BY marca
```

```
SELECT genero.nombre, AVG(duracion)
FROM peliculas
INNER JOIN generos ON generos.id = genero_id
GROUP BY genero.nombre
```

Módulo: MySQL
HAVING

HAVING

Cumple la misma función que WHERE, a diferencia de que HAVING se va a poder usar en conjunto con las **funciones de agregación** para filtrar **datos agregados**.

SQL

```
SELECT columna
FROM tabla
WHERE condicion
GROUP BY columna
HAVING condicion
ORDER BY columna;
```

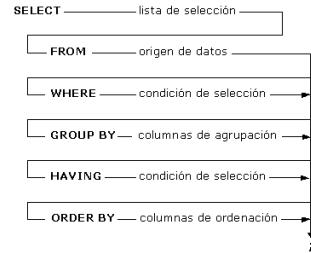
HAVING

Esta consulta devolverá la cantidad de clientes por país (agrupados por país). Solamente se incluirán en el resultado aquellos países que tengan **al menos** 3 clientes.

SQL

```
SELECT COUNT(cliente_id), pais
FROM clientes
GROUP BY pais
HAVING COUNT(clienteId) > 3;
```

ESTRUCTURA DE UNA QUERY



“

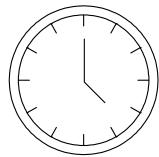
Las promesas son funciones que permiten ejecutar código asincrónico de forma eficiente.



PROMESAS

2

PROMESAS Pedido asincrónico



Un pedido asincrónico son instrucciones que se ejecutan mediante un mecanismo específico como por ejemplo un callback, una promesa o un evento. Lo que hace posible que la respuesta sea procesada en otro momento. Como se puede adivinar, su comportamiento es no bloqueante ya que el pedido se ejecuta en paralelo con el resto del código.

3

PROMESAS .then()

La función asincrónica devolverá un resultado, o no. Mientras tanto, el código se sigue ejecutando.

```
obtenerUsuarios()
  .then(function(data){
    console.log(data);
  });
  console.log("Se sigue ejecutando!")
```

Función asincrónica.
Código que podría seguirse ejecutando mientras la promesa se ejecuta.



4

PROMESAS .then()

Cuando la promesa se cumpla y se obtenga un resultado, entra en ejecución el primer .then() este actúa solo en consecuencia de la función asincrónica.

```
obtenerUsuarios()
  .then(function(data){
    console.log(data);
  });
  console.log("Se sigue ejecutando!")
```

Ejecuta el console.log() SÓLO SI obtenerUsuarios() devuelve un resultado. Este lo recibe .then() dentro de su callback, en este caso en el parámetro data.

5

PROMESAS Promesas anidadas

A veces los .then() suelen tener promesas dentro. Para resolver esto, necesitamos utilizar otro .then() que entre en ejecución una vez se resuelva el anterior.

```
obtenerUsuarios()
  .then(function(data){
    return filtrarDatos(data);
  })
  .then(function(dataFiltrada){
    console.log(dataFiltrada);
  })
```



6

¡ ATENCIÓN !



Es importante recordar que los `.then()` necesitan retornar la data procesada para que pueda ser usada por otro `.then()`.



PROMESAS

`.catch()`

En caso de NO obtener un resultado, se genera un error. Para esto usamos `.catch()`, que encapsula cualquier error que pueda generarse a través de las promesas. Dentro de este método decidimos qué hacer con el error. El mismo es recibido como parámetro dentro del callback del `.catch()`. En el siguiente ejemplo mostraremos el error en consola:

```
obtenerUsuarios()
  .then(function(data){
    console.log(data);
  })
  .catch(function(error){
    console.log(error);
  })
```



7

8

PROMESAS

Documentación

Para saber más sobre promesas, podés acceder a la documentación oficial de Mozilla haciendo click en el siguiente [Link](#):

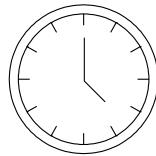


9

10

Promise.all()

Promise.all() Promesas en conjunto



A veces necesitamos que dos o más promesas se resuelvan para realizar cierta acción. Para esto usamos **Promise.all()**. Éste contendrá un array de promesas que una vez se hayan resuelto, un **.then()** se ejecutará con los resultados de las mismas.

2

PROMESAS

Promise.all()

Lo que primero debemos hacer es guardar en variables las promesas que necesitamos obtener.

```
let promesaPeliculas = obtenerPeliculas();           Promesa de películas.  
{}  
  
let promesaGeneros = obtenerGeneros();             Promesa de géneros.
```



3

PROMESAS

Promise.all()

El próximo paso es utilizar el método **Promise.all()** que contendrá un array con las promesas que guardamos anteriormente.

```
Promise.all([promesaPeliculas, promesaGeneros])
```

Promesa de películas.



4

PROMESAS

Promise.all()

El callback del **.then()** recibe un array con los resultados de las promesas cumplidas.

```
Promise.all([promesaPeliculas, promesaGeneros])  
  .then(function([resultadoPeliculas, resultadoGeneros]) {  
    console.log(resultadoPeliculas, resultadoGeneros);  
  })
```



El **.then()** se ejecutará solo si ambas promesas se cumplieron.

PROMESAS

Documentación

Para saber más sobre **Promise.all()**, podés acceder a la documentación oficial de Mozilla haciendo click en el siguiente [Link](#):



6

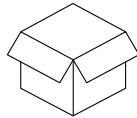
SEQUELIZE

Sequelize es un ORM que nos ayuda a conectarnos e interactuar con bases de datos como Postgres, MySQL, MariaDB, SQLite, Microsoft SQL Server y más.



SEQUELIZE Preparando el proyecto

Antes de instalar Sequelize, debemos tener en cuenta que al ser un paquete utilizado por NodeJS vamos a tener que utilizar NPM para esto.



3

SEQUELIZE Instalación

Dentro de la carpeta del proyecto de Node, hay que ejecutar los siguientes comandos:

```
>_ npm install sequelize-cli -g  
>_ npm install sequelize  
>_ npm install mysql2
```

SEQUELIZE Establecer rutas y directorios

Una vez instalados los paquetes que necesitamos, debemos crear un archivo llamado `.sequelizerc` en la raíz del proyecto y dentro dentro de este mismo escribir lo siguiente:



```
const path = require('path')  
  
module.exports = {  
  config: path.resolve('./database/config', 'config.js'),  
  'models-path': path.resolve('./database/models'),  
  'seeders-path': path.resolve('./database/seeders'),  
  'migrations-path': path.resolve('./database/migrations'),  
}
```

5

SEQUELIZE Iniciar Sequelize en tu proyecto

Para que Sequelize cree todas las carpetas y archivos que necesitamos para comenzar a trabajar con él, debemos correr el siguiente comando:

```
>_ sequelize init
```



2

4

6

SEQUELIZE

Configurando la conexión con la base de datos

Dentro de las carpetas que creó Sequelize encontraremos el archivo **config.js** en la ruta **/database/config/config.js**. Dentro de este encontramos un JSON con credenciales por defecto que debemos reemplazar por las nuestras.

```
{ }  
"development": {  
  "username": "root",  
  "password": "Monito123!",  
  "database": "movies_db",  
  "host": "127.0.0.1",  
  "dialect": "mysql",  
  "operatorsAliases": false  
}
```



7

¡ ATENCIÓN !

Necesitamos agregar un pequeño detalle para que no nos encontremos con un problema a la hora de requerir Sequelize. Para esto, en el archivo **config.js** que editamos anteriormente debemos **asignar** todo el JSON que modificamos a **module.exports**.



```
JS config.js x  
database > config > JS config.js > [o] <unknown>  
1 module.exports = [  
2   "development": {  
3     "username": "root",  
4     "password": "Monito123!",  
5     "database": "movies_db",  
6     "host": "127.0.0.1",  
7     "dialect": "mysql",  
8     "operatorsAliases": false  
9   },|
```

8

SEQUELIZE

Objeto DB

Como dato curioso les contamos que al final del archivo **index.js** ubicado en **/database/models/index.js** encontramos la exportación del objeto **DB**. Este será al que llamaremos cada vez que queramos utilizar Sequelize para realizar consultas a nuestra base de datos.



9

SEQUELIZE

Documentación

Para saber más puedes acceder a la documentación oficial de Sequelize haciendo click en el siguiente [Link](#):



10

“

SEQUELIZE Modelos



En los patrones de diseño MVC (Modelo - Vista - Controlador), los modelos contienen únicamente los datos puros de aplicación; no contiene lógica que describa cómo pueden presentarse los datos a un usuario. Este puede acceder a la capa de almacenamiento de datos. Lo ideal es que el modelo sea independiente del sistema de almacenamiento.

Es decir...

Un modelo es la representación de nuestra tabla en código. Con esto obtenemos recursos que nos permiten realizar consultas e interacciones con la base de datos de manera simplificada usando en este caso Sequelize.



3

peliculas.js

Pelicula.js

MODELOS Creando un modelo

Siempre creamos un modelo para cada tabla de nuestra base de datos. La ruta donde los almacenamos es /database/models.



- Los modelos son archivos JS, por lo tanto deben ser creados con esa extensión.
- Los nombres de los modelos deben estar escritos en UpperCamelCase y en singular.

4

MODELOS Creando un modelo

Un modelo es naturalmente una función que debemos definir y luego exportar con `module.export`. Esta función recibe dos parámetros. En primer lugar recibe el objeto `sequelize` para poder acceder a su método `define()` y en segundo lugar necesitamos traer al objeto `DataTypes` que nos dará la posibilidad de decirle a nuestras columnas qué tipo de datos permitirán.



Recorda que al estar tratándose de parámetros no es necesario que se llamen así pero solemos hacerlo para entender por qué los usamos.



5

6

MODELOS

Método define()

El método **define()** nos permite definir asignaciones entre un modelo y una tabla. Este recibe **3 parámetros**. El primero es un alias que identifica al modelo, el segundo es un objeto con la configuración de las columnas en la base de datos y el tercero es otro objeto con configuraciones adicionales (parámetro opcional). Lo que devuelva **define()** será almacenado en una variable con el nombre del modelo para luego ser retornada por la función que creamos.

```
js const Pelicula = sequelize.define(alias, cols, config);
return Pelicula;
```

7

MODELOS

Alias

Como nombramos en el slide anterior, el primero es un alias que utiliza Sequelize para **identificar al modelo**. No es algo determinante. Solemos asignarle el mismo nombre del modelo como **String**.

```
js const Pelicula = sequelize.define("Pelicula", cols,
config);
return Pelicula;
```

8

MODELOS

Tipos de datos en Sequelize

Dentro de nuestro segundo parámetro que llamamos **cols** se encuentra un objeto que nos permite, en el segundo parámetro del **define()**, definir qué tipos de datos deben recibir las columnas en la base de datos.

```
js cols = {
  id: {
    type: DataTypes.INTEGER
  },
  name: {
    type: DataTypes.STRING
  },
  admin: {
    type: DataTypes.BOOLEAN
  }
}
```

9

MODEL

Timestamps

```
module.exports = (sequelize, DataTypes) => {
  const Usuario = sequelize.define("Usuario", {
    email: {
      type: DataTypes.STRING
    },
    createdAt: {
      type: DataTypes.DATE
    },
    updatedAt: {
      type: DataTypes.DATE
    }
  });

  return Usuario;
}
```



Los timestamps no son obligatorios pero la mayoría de las tablas suelen tenerlos y forman parte del standard. Estos deben llamarse de la misma forma que se ve en el ejemplo.

Campos que guardan la fecha de creación y última edición.

11

MODELOS

Configuraciones adicionales

Dentro de nuestro tercer parámetro del **define()** podemos configurar cosas adicionales. Por ejemplo si el nombre de nuestra tabla está en inglés y el de nuestro modelo en español, deberíamos aclararle al modelo que esto es así mediante un objeto literal como en el ejemplo de la siguiente diapositiva.

12

```
js  module.exports = (sequelize, DataTypes) => {  
  const Pelicula = sequelize.define("Pelicula",  
    {  
      // Configuraciones de las columnas.  
    },  
    {  
      tableName: 'movies',  
      //Si el nombre de la tabla no coincide con  
      //el del modelo  
      timestamps: false,  
      //Si no tengo timestamps  
    });  
  
    return Pelicula;  
}
```

SEQUELIZE Documentación

Para más información acerca de DataTypes y qué cosas se pueden configurar de una misma columna en la DB ingresa al siguiente [Link](#).



SEQUELIZE SELECT

Sequelize utiliza una función llamada **FIND** para buscar información en una base de datos. Junto con **FIND** tenemos algunas variaciones como **findAll()**, **findOne()**, **findByPk()**.



SELECT findAll()

Para buscar todos los datos registrados en la tabla debemos usar **findAll()**.

```
js const db = require('../database/models');
db.Usuario.findAll()
  .then((resultados) => {
    console.log(resultados);
});
```



SELECT findAll()

Para buscar todos los datos registrados en la tabla debemos usar **findAll()**.

```
js const db = require('../database/models');
db.Usuario.findAll()
  .then((resultados) => {
    console.log(resultados);
});
```

Incluir la conexión a la base de datos.

3

4

SELECT findAll()

Para buscar todos los datos registrados en la tabla debemos usar **findAll()**.



```
js const db = require('../database/models');
db.Usuario.findAll()
  .then((resultados) => {
    console.log(resultados);
});
```

La función **findAll()** devuelve una **promesa**, por lo tanto, la usamos para usar el resultado de la búsqueda.

SELECT findAll()

Para buscar todos los datos registrados en la tabla debemos usar **findAll()**.

```
js const db = require('../database/models');
db.Usuario.findAll()
  .then((resultados) => {
    console.log(resultados);
});
```

El resultado se asignará en el **parámetro** de esta función, aquí lo llamamos **resultados**, pero podría ser cualquier otro nombre.

5

6

SELECT

findOne()

`findOne()` permite que busquemos resultados que coincidan con los atributos indicados en el objeto literal que recibe el método.

```
js   db.Usuario.findOne({  
    where: {  
      nome: 'Tony'  
    }  
}).then((resultado) =>{  
  console.log(resultado);  
});
```



SELECT

findById()

El método `findById()` busca un registro con la clave primaria del mismo valor al parámetro pasado:

7

```
js   db.Auto.findById(42)  
      .then((resultado) => {  
        console.log(resultado);  
      });  
  
// SELECT * FROM autos WHERE id = 42;
```



SEQUELIZE

Para saber más

Para obtener más información, visite la documentación oficial haciendo click en el siguiente [Link](#).



9

“

SEQUELIZE WHERE

A menudo queremos buscar en la base de datos, pero no queremos todos los registros, solo aquellos que cumplan una condición. Para filtrar datos usamos un objeto literal con el atributo **WHERE** y un método de búsqueda.



SEQUELIZE WHERE

Para agregar una condición a la consulta, simplemente pase el atributo **where** al método **findAll()**:

```
js
const db = require('../database/models');

db.Auto.findAll({
  where: {
    marca: 'Fiat'
  }
}).then(resultados=>{
  console.log(resultados);
})
```



SEQUELIZE WHERE

Para agregar una condición a la consulta, simplemente pase el atributo **where** al método **findAll()**:

```
js
const db = require('../database/models');

db.Auto.findAll({
  where: {
    marca: 'Fiat'
  }
}).then(resultados=>{
  console.log(resultados);
})
```

Dentro del **where** pasamos el atributo de acuerdo con la columna de la tabla y el valor a buscar

3

4

SEQUELIZE WHERE

Para obtener más información, visite la documentación oficial haciendo click en el siguiente [Link](#):



5

SEQUELIZE ORDER, OFFSET Y LIMIT

“

ORDER es una forma de ordenar el resultado de la consulta a la base de datos a través de una columna elegida.

Puedo ordenar los elementos por id, fecha de creación, nombre, etc.



ORDENACION ORDER

En Sequelize, para ordenar el resultado simplemente hay que usar el atributo **order** que recibe un array:

```
js db.Usuario.findAll({  
    order: [  
        ['nombre', 'ASC'],  
    ],  
});
```

ORDENACION ORDER

En Sequelize, para ordenar el resultado simplemente hay que usar el atributo **order** que recibe un array:

```
js db.Usuario.findAll({  
    order: [  
        ['nombre', 'ASC'],  
    ],  
});
```

El primer valor del array es la columna que desea ordenar, el segundo valor es el orden de ordenamiento: ascendente o descendente.

3

4

“

LIMIT sirve para limitar el número de resultados a obtener.



ORDENACION LIMIT

Para limitar el número de resultados, simplemente hay que agregar el atributo **limit** al objeto y pasarlo al **findAll()**:

```
js db.Usuario.findAll({  
    limit: 10  
}).then((resultados) => {  
    console.log(resultados)  
})
```

5

6

OFFSET es sirve para omitir varios resultados, ampliamente utilizado para paginar los resultados.



ORDENACION OFFSET

Para omitir varios elementos de resultado, simplemente hay que agregar el atributo **offset** al objeto y pasarlo al **findAll()**:

```
db.Usuario.findAll({  
  offset: 10  
}).then((resultados) => {  
  console.log(resultados)  
})
```

ORDENACION ORDER + LIMIT + OFFSET

Se pueden usar todos los métodos juntos simplemente pasándolos como parámetros al **findAll()**:

```
db.Usuario.findAll({  
  order: [  
    ['nome', 'ASC'],  
  ],  
  offset: 5,  
  limit: 10  
})
```

SEQUELIZE ORDER, OFFSET Y LIMIT

Para obtener más información, visite la documentación oficial haciendo click en el siguiente [Link](#).



SEQUELIZE

FUNCIONES DE AGREGACIÓN

“

Sequelize también ofrece **funciones de agregación** para traer datos como suma, redondeo, promedio, mínimo y máximo.



FUNCIONES DE AGREGACIÓN

COUNT()

El método `count()` devuelve el total de resultados obtenidos con tu búsqueda:

```
js db.Usuario.count().then(resultado => {
  console.log(`Tenemos ${resultado} usuarios!`)
```



FUNCIONES DE AGREGACIÓN

COUNT() + WHERE()

Usando `count()` con `where` obtenemos el número de registros con la condición asignada en el `where`:

```
js db.Usuario.count({
  where: {
    admin: true
  }
}).then(resultado => {
  console.log(`Tenemos ${resultado} administradores!`)
```

FUNCIONES DE AGREGACIÓN

MAX()

La función `max()` trae el valor máximo de un atributo, en el ejemplo estamos filtrando el valor máximo de la columna 'edad':

```
js db.Usuario.max('edad').then(resultado => {
  console.log(`El usuario más grande tiene ${resultado} años!`)
```



FUNCIONES DE AGREGACIÓN

SUM()

La función `sum()` trae la suma de todos los valores de una columna asignada por parámetro:

```
js db.Usuario.sum('saldo').then(resultado => {
  console.log(`La suma de saldos de todos los usuarios es
  ${resultado}`)
```

5

2

4

6

SEQUELIZE **FUNCIONES DE** **AGREGACIÓN**

Para obtener más información, visite la documentación oficial haciendo click en el siguiente [Link](#):



SEQUELIZE RAW QUERIES

Las consultas sin formato o Raw Queries son una forma de manipular la base de datos con Sequelize mediante consultas SQL.



SEQUELIZE Preparando el proyecto

Asegúrese de que su base de datos tenga la conexión configurada antes de usar raw queries con Sequelize.



3

SEQUELIZE Requerir el objeto DB



Lo único que necesitamos para poder utilizar Sequelize en nuestros controladores es requerir el objeto DB que exporta `/database/models/index.js`. A través de él accederemos a todos nuestros modelos.

js

```
let db = require("../database/models");
```

SEQUELIZE sequelize.query()

La función `query()` de sequelize recibe una consulta SQL como parámetro y se debe acceder al objeto `sequelize` para poder llegar a ella:

```
js
let db = require("../database/models");
db.sequelize.query("SELECT * FROM ...");
```



5

SEQUELIZE sequelize.query()

El retorno de `sequelize.query()` será un array de dos posiciones donde nuestros resultados estarán en la primer posición. Con esto, podemos definir qué hacer con esta información dentro del `callback` que recibe `.then()`. En el siguiente ejemplo, imprimimos el resultado de la consulta en la consola:

```
js
db.sequelize.query('SELECT * FROM usuarios')
  .then(resultados => {
    let usuarios = resultados[0];
    console.log(usuarios);
 });
```



6

SEQUELIZE

Documentación

La documentación de sequelize tiene ejemplos y explicaciones de varias formas de usar `sequelize.query()`. Para acceder a ella haz click en el siguiente [Link](#).



SEQUELIZE Create

Create es un método que nos permite agregar nuevos registros en nuestras tablas de la base de datos.



SEQUELIZE Método .create()

.create() es un método que le pertenece a los modelos de nuestra base de datos. Por lo tanto, para acceder a él necesitamos llamar primero al modelo.

```
js const db = require('../database/models');  
db.Usuario.create();
```

Como necesitamos llamar al modelo, debemos requerir la conexión a la DB

SEQUELIZE Definiendo el método

.create() recibe un objeto literal donde definimos qué campos vamos a modificar y con qué valores.

```
const db = require('../database/models');  
db.Usuario.create({  
  name: "Manuel",  
  username: "manolito",  
  password: "manolo123"  
});
```

El objeto literal que recibe .create() debe contener como KEY el mismo nombre del campo a escribir en la DB y como valor el contenido que queremos almacenar en él.

3

4

SEQUELIZE Documentación

Para saber más puedes acceder a la documentación oficial de Sequelize haciendo click en el siguiente [Link](#):



5

SEQUELIZE

Update

Update es un método que nos permite editar registros en nuestras tablas de la base de datos.



SEQUELIZE

Método .update()

.update() es un método que le pertenece a los modelos de nuestra base de datos. Por lo tanto, para acceder a él necesitamos llamar primero al modelo.

```
js
const db = require('../database/models');
db.Usuario.update();
```

Como necesitamos llamar al modelo, debemos requerir la conexión a la DB

3

SEQUELIZE

Conceptos importantes

.update() recibe dos parámetros. Ambos son objetos literales. En el primero debemos indicarle qué campo de la tabla modificar y qué valor reasignarle. El segundo objeto debe tener como mínimo un where que indique de manera única a qué registro aplicar los cambios. En caso de no hacerlo se modificarán todos los campos de la DB.



IMPORTANTE! No olvidar el WHERE

SEQUELIZE

Definiendo el método

El primer parámetro de .update() recibe un objeto literal donde indicamos qué campo de la base de datos modificar y el valor que queremos asignarle.

```
js
const db = require('../database/models');

db.Usuario.update({
  username: 'ManuelF'
},
{
  where: {
    id: 10
  }
});
```

Modificamos el campo "username" para que ahora contenga "ManuelF"

js

5

SEQUELIZE

Definiendo el método

En el segundo parámetro le indicamos al .update() qué fila queremos modificar. Para esto usamos un where, donde mediante **campos únicos** nos referimos a qué registro modificar.

```
js
const db = require('../database/models');

db.Usuario.update({
  username: 'ManuelF'
},
{
  where: {
    id: 10
  }
});
```

Modificamos el registro que tenga ID = 10

6

SEQUELIZE

Documentación

Para saber más puedes acceder a la documentación oficial de Sequelize haciendo click en el siguiente [Link](#)



SEQUELIZE Destroy

Destroy es un método que nos permite eliminar registros en nuestras tablas de la base de datos.



SEQUELIZE Método .destroy()

.destroy() es un método que le pertenece a los modelos de nuestra base de datos. Por lo tanto, para acceder a él necesitamos llamar primero al modelo.

```
js const db = require('../database/models');  
db.Usuario.destroy();
```

Como necesitamos llamar al modelo, debemos requerir la conexión a la DB

3

SEQUELIZE Conceptos importantes

.destroy() recibe un solo parámetro. Éste será un objeto con un where que tendrá la condición que necesitamos aclarar para eliminar el registro que nosotros queramos. En caso de no hacerlo se eliminarán todos los campos de la DB.



IMPORTANTE! No olvidar el WHERE

SEQUELIZE Definiendo el método

.destroy() recibe un objeto literal donde indicaremos qué registro queremos eliminar. Para esto necesitamos un where.

```
js const db = require('../database/models');  
db.Usuario.destroy({  
  where: {  
    id: 10  
  }  
});
```

Eliminamos el registro que tenga ID = 10

5

SEQUELIZE Documentación

Para saber más puedes acceder a la documentación oficial de Sequelize haciendo click en el siguiente [Link](#)



SEQUELIZE RELACIONES 1:N

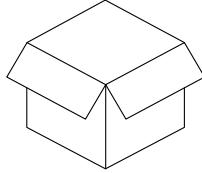
Las relaciones en Sequelize existen para optimizar la obtención de datos en una consulta a la base de datos.



SEQUELIZE .associate

Las relaciones se aplican dentro de nuestro modelo de tabla. Luego de definirlo con el método `.define()` debemos llamar a la variable creada y utilizar la propiedad `.associate` para definir nuestras relaciones.

`.associate` debe almacenar una función anónima que recibe un solo parámetro. Este será un **objeto que contiene todos tus modelos**, pudiendo acceder así a cada uno de ellos.



js

SEQUELIZE .associate

```
const Pelicula = sequelize.define(alias, config, configAd);
Pelicula.associate = function(modelos){
  // Relación
}
```

Definimos una función en la propiedad `.associate` de nuestra variable que representa al modelo (`Pelicula`).

3

4

SEQUELIZE .belongsTo() | .hasMany()

`.belongsTo()` y `.hasMany()` son métodos de Sequelize que nos permiten indicarle a nuestros modelos la relación de estos con otros. Estos se declaran dentro de la función que asignamos a `.associate` y deben aplicarse a nuestra variable definida con `.define()`.

```
const Pelicula = sequelize.define(alias, config, configAd);
Pelicula.associate = function(modelos){
  Pelicula.belongsTo();
  Pelicula.hasMany();
}
```

Creamos las relaciones dentro de la función asignada a `.associate`.

5

SEQUELIZE .belongsTo()

`.belongsTo()` lo utilizamos para decir que un registro puede estar asociado a uno o más de otra tabla. En este caso la relación sería de 1:N.

Para definir la relación, `.belongsTo()` recibe dos parámetros, donde el primero es el modelo con que querés relacionarlo (Llamándolo a través del parámetro que contiene nuestros modelos) y el segundo es un objeto donde debemos detallar la relación.

6

SEQUELIZE .belongsTo()

```
js Pelicula.associate = function(modelos){  
  Pelicula.belongsTo(modelos.Generos, {  
    as: "generos",  
    foreignKey: "genre_id"  
  });
```

Asignamos un alias con el que llamaremos luego a la relación.

SEQUELIZE .belongsTo()

```
js Pelicula.associate = function(modelos){  
  Pelicula.belongsTo(modelos.Generos, {  
    as: "generos",  
    foreignKey: "genre_id"  
  });
```

Aclaramos la foreignKey donde se relacionan ambas tablas.

7

8

SEQUELIZE .hasMany()

.hasMany() lo utilizamos para decir que uno o más registros pueden estar asociados a uno solo de otra tabla. En este caso la relación sería de N:1.

Para definir la relación, .hasMany() recibe dos parámetros, donde el primero es al modelo con que querés relacionarlo (llamándolo a través del parámetro que contiene nuestros modelos) y el segundo es un objeto donde debemos detallar la relación.

SEQUELIZE .hasMany()

```
js Genero.associate = function(modelos){  
  Genero.belongsTo(modelos.Pelicula, {  
    as: "peliculas",  
    foreignKey: "genre_id"  
  });
```

Asignamos un alias con el que llamaremos luego a la relación.

9

10

SEQUELIZE .hasMany()

```
js Genero.associate = function(modelos){  
  Genero.belongsTo(modelos.Pelicula, {  
    as: "peliculas",  
    foreignKey: "genre_id"  
  });
```

Aclaramos la foreignKey donde se relacionan ambas tablas.

11

SEQUELIZE Conceptos importantes



¡ATENCIÓN!

Siempre que creamos una relación desde un modelo, debemos generar la misma desde el otro con quien está relacionado. Si no lo hacemos, Sequelize no reconoce la asociación.

12

SEQUELIZE

Documentación

Para saber más puedes acceder a la documentación oficial de Sequelize haciendo click en el siguiente [Link](#)



SEQUELIZE RELACIONES N:M

Las relaciones en Sequelize existen para optimizar la obtención de datos en una consulta a la base de datos.



SEQUELIZE .belongsToMany()

Hay veces donde una tabla posee una relación de muchos a muchos. Es decir, muchos registros de nuestra tabla pueden estar relacionados con varios de otra. Para poder aclararle esta relación a Sequelize debemos utilizar el método `.belongsToMany()`.

Para definir la relación, `.belongsToMany()` recibe dos parámetros, donde el primero es el modelo con que querés relacionarlo (Llamándolo a través del parámetro que contiene nuestros modelos) y el segundo es un objeto donde debemos detallar la relación.

SEQUELIZE .belongsToMany()

```
Pelicula.associate = function(modelos){
  Pelicula.belongsToMany(modelos.Actores, {
    as: "actores",
    through: "actor_movie",
    foreignKey: "genre_id",
    otherKey: "actor_id",
    timestamps: false
  });
}
```

Asignamos un alias con el que llamaremos luego a la relación.

SEQUELIZE .belongsToMany()

```
Pelicula.associate = function(modelos){
  Pelicula.belongsToMany(modelos.Actores, {
    as: "generos",
    through: "actor_movie",
    foreignKey: "genre_id",
    otherKey: "actor_id",
    timestamps: false
  });
}
```

Nombre de la tabla intermedia que utilizamos para la relación

SEQUELIZE .belongsToMany()

```
Pelicula.associate = function(modelos){
  Pelicula.belongsToMany(modelos.Actores, {
    as: "generos",
    through: "actor_movie",
    foreignKey: "genre_id",
    otherKey: "actor_id",
    timestamps: false
  });
}
```

Referencia a la tabla del modelo que estamos referenciando

SEQUELIZE .belongsToMany()

```
js Pelicula.associate = function(modelos){  
  Pelicula.belongsToMany(modelos.Actores, {  
    as: "generos",  
    through: "actor_movie",  
    foreignKey: "genre_id",  
    otherKey: "actor_id",  
    timestamps: false  
  });  
}
```

Referencia a la otra tabla que queremos referenciar

SEQUELIZE .belongsToMany()

```
js Pelicula.associate = function(modelos){  
  Pelicula.belongsToMany(modelos.Actores, {  
    as: "generos",  
    through: "actor_movie",  
    foreignKey: "genre_id",  
    otherKey: "actor_id",  
    timestamps: false  
  });  
}
```

Aclaramos si la tabla pivot posee o no timestamps.

7

8

SEQUELIZE Conceptos importantes



¡ATENCIÓN!

Siempre que creamos una relación desde un modelo, debemos generar la misma desde el otro con quien está relacionado. Si no lo hacemos, Sequelize no reconoce la asociación.

9

SEQUELIZE Documentación

Para saber más puedes acceder a la documentación oficial de Sequelize haciendo click en el siguiente [Link](#)

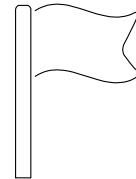


10

SEQUELIZE RELACIONES

SEQUELIZE Requerimientos

Para poder utilizar las relaciones necesitamos corroborar que estas estén correctamente definidas en los modelos de nuestras tablas.



SEQUELIZE Implementación

Si recordamos, nuestros métodos que ejecutan consultas a la DB suelen recibir un objeto como parámetro. Dentro de este deberíamos agregar un atributo llamado **include** que contenga un array de objetos.

Cada objeto de ese array representará una relación. Si no aclaramos cuál queremos usar, cuando queramos usarla en otro lugar no existirá.

Para definir cada relación debemos crear un atributo llamado **association** que contenga el mismo alias que usamos en el modelo para llamar a la relación.

SEQUELIZE Implementación

```
// Controlador
const db = require('../database/models');

db.Peliculas.findAll({
  include: [
    {association: "genero"}, 
    {association: "actores"} 
  ]
}).then(resultados=>{
  console.log(resultados);
})
```

Incluimos las asociaciones (relaciones) que creamos en los modelos.

SEQUELIZE En la vista

Para finalizar, nos queda poder utilizar esta información en la vista. Para esto debemos acceder a las asociaciones mediante atributos del modelo. Estos poseen el mismo nombre del alias creado en la definición de la relación.



```
// Muestro el género al que pertenece la película
<%= pelicula.genero %>

// Muestro los actores de la película
<% for (let i = 0; i < pelicula.actores.length; i++) { %>
  <%= pelicula.actores[i] %>
<% } %>
```

VINCULACIÓN

VINCULACIÓN INTERNA

```
<body>
...
<script>
    console.log("Hola Mundo!");
</script>
</body>
```

Nos permite escribir código js directamente en nuestro archivo HTML.

NO ES LO MAS PROLIJO!

VINCULACIÓN EXTERNA

```
<body>
...
<script src="js/main.js"></script>
</body>
```

Nos permite linkear nuestro archivo HTML con un archivo JS externo.

VINCULACIÓN EXTERNA

```
let saludo = "Hola mundo!";
console.log(saludo);
```

Recordá que con el uso de la vinculación externa no es necesario dentro de nuestro archivo con extensión .js escribir las etiquetas <script>

“

Una buena práctica es poner la vinculación con nuestro archivo js **al final** del body de nuestro HTML de manera que el documento ya esté cargado para cuando se lea el script.



DOCUMENT OBJECT MODEL (D.O.M.)

¿Qué nos permite hacer JS con el D.O.M.?

- Modificar elementos, atributos y estilos de una página.
- Borrar cualquier elemento y atributo.
- Agregar nuevos elementos o atributos.
- Reaccionar a todos los eventos HTML de la página.
- Crear nuevos eventos HTML en la página.

OBJETO WINDOW

DigitalHouse>
Coding School



OBJETO WINDOW

OBJETO WINDOW

El objeto **window** es lo **primero** que se carga en el navegador.

Representa **la ventana donde estamos navegando** y nos da la interfaz para operar con el navegador.

El objeto window tiene propiedades como **length**, **innerWidth**, etc.

DigitalHouse>
Coding School

OBJETO DOCUMENT



OBJETO DOCUMENT

Representa al **HTML** y nos va a dar una interfaz, un conjunto de atributos y de métodos para poder efectivamente leer lo que tenemos en el HTML y si quisieramos modificarlo.

El document es cargado dentro del objeto **window** y tiene propiedades como title, url, cookie, etc.

“SELECTORES”

“

Para acceder a los elementos de una página, usamos **selectores**. Cada selector puede retornar **un solo elemento o una lista de elementos**.



“

Para poder hacer uso de los selectores debemos hacer uso del objeto **document** ya que los selectores son métodos del mismo.



querySelector();

Este selector recibe un String que indica el selector CSS del elemento de D.O.M. que estamos buscando.

Por ejemplo:

```
let titulo = document.querySelector(".title");
```

Nos va a retornar el **primer** elemento del HTML que contengan la clase “title”.



Es importante declarar una variable para almacenar el dato que nos traiga el selector, ya que de otra manera lo perderíamos al continuar la ejecución del programa.

querySelectorAll();

Este selector recibe un String que indica el selector CSS del elemento de D.O.M que estamos buscando.

Por ejemplo:

```
let nombres=document.querySelectorAll(".name");
```

Nos va a retornar **un listado** de elementos que coincidan con la búsqueda especificada.



También podemos utilizar los selectores directamente con elementos del documento, por ejemplo:

```
let divs = document.querySelectorAll("div");
```

getElementById();

Este selector recibe un String con únicamente el nombre del id del elemento de D.O.M que estamos buscando.

Por ejemplo:

```
let marca= document.getElementById("marca");
```

Nos va a retornar el **elemento cuyo id coincide con el deseado**.



También podemos buscar elementos por su id mediante los selectores anteriores, pero debemos anteponerle un # para aclarar que es un id.

```
let marca = document.querySelector("#marca");
```

MODIFICANDO EL D.O.M.

“

Para poder hacer modificaciones al D.O.M. siempre tenemos que tener seleccionado el objeto que queremos modificar. Esto lo podemos hacer usando selectores!



innerHTML

Si queremos **leer o modificar** el contenido de una etiqueta HTML vamos a utilizar esta propiedad.

```
{ } document.querySelector("div.nombre").innerHTML;
```

En este caso la utilizamos para leer el contenido. Pero ¿Y si queremos modificarlo?



Si queremos **guardarnos** el valor debemos asignar esa línea de código a una variable. De otra manera cuando la ejecución continúe se perderá el valor que hayamos buscado.

3

innerHTML

Si no queremos perder el contenido que tenemos en la etiqueta a la que le queremos hacer cambios harímos:

```
{ } document.querySelector("compras").innerHTML += "Papitas";
```

De esta forma estamos agregando al div con clase compras, la palabra “Papitas” de tal manera que si lo leyéramos diría

```
<div class="compras"> Jamón, Queso, Pan Papitas</div>
```

Contenido que ya estaba

Contenido que agregué

5

innerHTML

Si queremos **modificar** el contenido de una etiqueta HTML vamos a utilizar esta propiedad.

```
{ } document.querySelector("div.nombre").innerHTML = "Darío";
```

Si utilizamos la propiedad de esta manera, todo el contenido que teníamos en el div con clase nombre, se va a cambiar por el string “Darío”.



Sin embargo, también podríamos modificar el contenido sin perder lo que teníamos anteriormente...

4

innerText

Si queremos **leer o modificar** el texto que posea un elemento HTML, podemos usar esta propiedad de la siguiente manera:

```
{ } document.querySelector("div.nombre").innerText;
```

En este caso si en mi div con clase nombre, estuviera escrito “Leo”, la propiedad me retornaría el **String** “Leo”.



Si queremos **guardarnos** el valor debemos asignar esa línea de código a una variable. De otra manera cuando la ejecución continúe se perderá el valor que hayamos buscado.

6

innerText

Si queremos **modificar** el texto que posea un elemento HTML podemos usar esta propiedad de la siguiente manera:

```
{ document.querySelector("div.nombre").innerText += "Messi"; }
```

En este caso lo que sucedería es similar a lo que sucede con el otro selector pero el texto se incluiría dentro de la etiqueta div, quedando:

```
<div class="nombre">Leo Messi</div>
Contenido que ya estaba
↓
Contenido que agregué
```

MODIFICANDO ESTILOS

“

Además de agregar y modificar elementos HTML, con los selectores también podemos **modificar los estilos** de un elemento HTML



PROPIEDAD **STYLE**

Nos permite leer y sobreescribir las reglas CSS que se aplican sobre un elemento que hayamos seleccionado.

```
let titulo = document.querySelector(".title");
titulo.style.color = "cyan";
titulo.style.textAlign = "center";
titulo.style.fontSize = "12px";
titulo.style.backgroundColor = "#dddddd";
```



Nótese que las reglas CSS que llevaban guiones como `font-size`, en Javascript se escriben en camelCase es decir `fontSize`.

MODIFICANDO CLASES

“

Javascript nos da una propiedad y varios métodos que nos permiten hacer diversas acciones con el atributo class de un elemento



classList.add()

Nos permite agregar al elemento que tengamos seleccionado, una clase nueva.

```
0 let cita = document.querySelector(".cita");
cita.classList.add("italicas");
```

antes

```
▶ html css
<p class="cita">el
veláz murciélagos comía
feliz cardillo y
kiwi</p>
```

después

```
▶ html css
<p class="cita
italicas">el veláz
murciélagos comía feliz
cardillo y kiwi</p>
```

3

classList.remove()

Nos permite quitarle una clase existente al elemento que tengamos seleccionado.

```
0 let cita = document.querySelector(".cita");
cita.classList.remove("cita");
```

antes

```
▶ html css
<p class="cita
italicas">el veláz
murciélagos comía feliz
cardillo y kiwi</p>
```

después

```
▶ html css
<p class="italicas">el
veláz murciélagos comía
feliz cardillo y
kiwi</p>
```

4

classList.toggle()

Revisa si existe una clase en el elemento seleccionado, de ser así la remueve, de lo contrario, si la clase no existe, la agrega.

```
0 let cita = document.querySelector("p");
cita.classList.toggle("cita");
```

antes

```
▶ html css
<p class="italicas">el
veláz murciélagos comía
feliz cardillo y
kiwi</p>
```

después

```
▶ html css
<p class="italicas
cita">el veláz
murciélagos comía feliz
cardillo y kiwi</p>
```

5

classList.contains()

Nos permite preguntar si un elemento tiene una clase determinada. Devuelve un valor booleano.

```
▶
let cita = document.querySelector(".italicas");
cita.classList.contains("cita"); // false;
```

```
▶
let cita = document.querySelector(".italicas");
cita.classList.contains("italicas"); // true;
```

6

“

Podemos usar el **.contains** para hacer operaciones lógicas haciendo uso de los **if / else**.



EVENTOS

“

Un **evento** es algo que pasa en el browser o que es ejecutado por el usuario. Por ejemplo:

- La página terminó de cargar.
- El usuario hizo click en un botón.
- El input del formulario cambió.



EVENTOS MÁS USADOS

- onclick
- ondblclick
- onmouseover
- onmouseout
- onmousemove
- onscroll
- onkeydown
- onload
- onfocus
- onblur
- onchange
- onsubmit

1

EVENTO ONLOAD

3

ONLOAD

El evento **onLoad** es un evento que permite que todo el script se ejecute cuando se haya cargado por completo el objeto **document** dentro del objeto **window**.

```
window.onload = function(){  
    console.log("el documento está listo");  
}
```



Se suele escribir el código js dentro de esta función para **prevenir errores** que pueden ocurrir si el documento no está totalmente cargado al momento de la ejecución del script.

5

2

EVENTO ONCLICK

ONCLICK

El evento **onclick** nos permite ejecutar una acción cuando se haga click sobre el elemento al cual le estamos aplicando la propiedad.

```
{}  
btn.onclick = function(){  
    console.log("hiciste click!");  
}  
}
```

3

PREVENT DEFAULT()

7

PREVENT DEFAULT ()

El **preventDefault()** nos permite evitar que se ejecute el evento predeterminado o nativo del elemento al que se lo estemos aplicando.

Podemos usarlo, por ejemplo, para prevenir que una etiqueta "a" se comporte de manera nativa y que en vez de eso haga otra acción.



Siempre tenemos que tener seleccionado el elemento al que le queremos aplicar el preventDefault() mediante los selectores.

PREVENT DEFAULT ()

```
{}  
let hipervinculo = document.querySelector("a");  
  
hipervinculo.addEventListener("click", function(event){  
    console.log("hiciste click");  
    event.preventDefault();  
});
```

9

10

PREVENT DEFAULT ()

```
{}  
let hipervinculo = document.querySelector("a");  
  
hipervinculo.addEventListener("click", function(event){  
    console.log("hiciste click");  
    event.preventDefault();  
});
```

Atrapamos el evento

Prevenimos la acción nativa

11

EVENTOS DE MOUSE

DOUBLECLICK

```
let texto = document.querySelector(".text");

() texto.ondblclick = function(){
    console.log("hiciste doble click");
}
```

También podríamos hacer...

```
let texto = document.querySelector(".text");

() texto.addEventListener("dblclick", function(){
    console.log("hiciste doble click");
});
```

MOUSEOVER

```
let texto = document.querySelector(".text");

() texto.onmouseover = function(){
    console.log("pasaste el mouse");
}
```

También podríamos hacer...

```
let texto = document.querySelector(".text");

() texto.addEventListener("mouseover", function(){
    console.log("pasaste el mouse");
});
```

MOUSEOUT

```
let texto = document.querySelector(".text");

() texto.onmouseout = function(){
    console.log("quitaste el mouse");
}
```

También podríamos hacer...

```
let texto = document.querySelector(".text");

() texto.addEventListener("mouseout", function(){
    console.log("quitaste el mouse");
});
```

“

Como habrás podido notar, aplicar el evento con un **.on"evento"** o con un **addEventListener("evento", function{...}** etc, no tiene diferencia!



Eventos de Teclado

onkeydown

El evento **keydown** es lanzado cuando una tecla es presionada (hacia abajo). A diferencia del evento keypress, el evento **keydown** es lanzado para las teclas que producen un carácter y también para las que no lo producen.

```
let miInput = document.querySelector('#miInput');
miInput.onkeydown = function(event){
    alert("Se presiono la tecla: " + event.key);
}
```

onkeyup

El evento es iniciado cuando la tecla es soltada.

```
let miInput = document.querySelector('#miInput');
miInput.onkeyup = function(event){
    alert("Se soltó la tecla: " + event.key);
}
```

onkeypress

Se dispara al finalizar el recorrido completo de presión y liberación de la tecla.

```
let miInput = document.querySelector('#miInput');
miInput.onkeypress = function(event){
    alert("Se presiono la tecla: " + event.key);
}
```

Timers

setTimeout

Se utiliza cuando queremos que nuestro código se ejecute una sola vez, pasado un tiempo determinado.

```
const delay = 3000;
setTimeout(miFuncion,delay);
function miFuncion(){
    alert("¡Este es mi anuncio!")
}
```

Nota

miFuncion, es aquella función la cual voy a ejecutar por callback cuando se cumpla el tiempo determinado por **delay** (en milisegundos, en el ejemplo serían: 3 segundos).

setInterval

Se utiliza cuando queremos que nuestro código se ejecute una y otra vez, pasado un tiempo determinado.

```
const delay = 3000;
setInterval(miFuncion,delay);
function miFuncion(){
    alert("¡Este es mi anuncio repetido : ) !")
}
```

Nota

miFuncion, es aquella función la cual voy a ejecutar por callback cuando se cumpla el tiempo determinado por **delay** (en milisegundos, en el ejemplo serían: 3 segundos).

clearTimeout y clearInterval

Ahora bien... ¿Puedo **detener** la ejecución de un timer cuando desee?

Obviamente! Tanto setTimeout como setInterval, pueden detenerse. Veamos el siguiente ejemplo con **clearInterval**:

```
const delay = 3000;
const miIntervalo = setInterval(miFuncion,delay);
function miFuncion(){
    alert("¡Este es mi anuncio repetido : ) !")
}
clearInterval(miIntervalo);
```

3

2

4

Eventos de Formulario

onfocus

El evento “**focus**” sucede cuando el usuario ingresa con el cursor dentro de un campo input.

```
let miInput = document.querySelector('#miInput');
miInput.onfocus = function(){
    alert('Ingresaste el cursor al campo :)');
}
```

onblur

Por el contrario el evento “**blur**” sucede cuando el cursor abandona el campo en donde se encuentra.

```
let miInput = document.querySelector('#miInput');
miInput.onblur = function(){
    alert('Saliste del campo :)');
}
```

onchange

El evento “**change**” permite identificar que el valor de un campo cambió. La ventaja que presenta el evento “**change**” es que podemos aplicarlo sobre cualquiera de los campos del formulario, inclusive sobre el formulario completo.

```
let miInput = document.querySelector('#miInput');
miInput.onchange = function(){
    alert('Hubo un cambio por aquí...');
}
```

onsubmit

El evento “**submit**” identifica el momento en que se cliquea sobre un botón o un campo input, ambos de tipo “**submit**”.

```
let miForm = document.querySelector('#miForm');
miForm.onsubmit = function(){
    alert('El formulario se está por enviar...');
}
```

onsubmit y preventDefault()

Para evitar el envío de un formulario, necesitamos incluir a la función **preventDefault** en la primera línea dentro de la función.

```
let miForm = document.querySelector('#miForm');
miForm.onsubmit = function(event){
    if( algunCampoEnBlanco ) {
        event.preventDefault();
        alert('El formulario no se envió...');
    }else{
        alert('El formulario se está por enviar...');
    }
}
```

setInterval

Se utiliza cuando queremos que nuestro código se ejecute una y otra vez, pasado un tiempo determinado.

```
const delay = 3000;
setInterval(miFuncion,delay);
function miFuncion(){
    alert("¡Este es mi anuncio repetido :) !")
}
```

Nota

miFuncion, es aquella función la cual voy a ejecutar por callback cuando se cumpla el tiempo determinado por **delay** (en milisegundos, en el ejemplo serían: 3 segundos).

clearTimeout y clearInterval

Ahora bien... ¿Puedo **detener** la ejecución de un timer cuando desee?

iObviamente! Tanto setTimeout como setInterval, pueden detenerse. Veamos el siguiente ejemplo con **clearInterval**:

```
const delay = 3000;
const miIntervalo = setInterval(miFuncion,delay);
function miFuncion(){
    alert("¡Este es mi anuncio repetido :) !")
}
clearInterval(miIntervalo);
```

VALIDACIONES

Obtener el formulario

El primer objetivo, será obtener el formulario y luego agregar un comportamiento cuando queramos capturar el evento.

```
window.addEventListener("load", function(){
  let formulario = document.querySelector("form.reservation")
  //Agregamos el evento para atraparlo
  formulario.addEventListener("submit", function(event){
    //Form recien atrapado :
  });
});
```

Validando un campo vacío

Primero y principal, dentro del evento, detendremos el envío de formulario con `event.preventDefault()`.

Luego, obtendremos nuestro `input` con `querySelector` para que finalmente preguntemos si el campo está vacío.

```
event.preventDefault();
let campoNombre = document.querySelector("input.nombre");
if(campoNombre.value == ""){
  alert("El campo nombre no debe estar vacío");
}
```

Array de errores

Nuestro siguiente paso es crear un **array** para acumular estos errores y cambiar nuestra lógica, es decir, si el array no está vacío, entonces prevengo el envío de formulario, caso contrario, el formulario se enviará.

```
let errores = [];
let campoNombre = document.querySelector("input.nombre");
if(campoNombre.value == ""){
  errores.push("El campo nombre está vacío");
}
if(errores.length > 0){
  event.preventDefault();
}
```

Mostrando errores en HTML

Nuestro siguiente paso es crear una etiqueta `<div>` dentro de nuestro html donde mostraremos el listado de errores. Para ello, la colocaremos por arriba de nuestro formulario sumándole una etiqueta `` para indicar el inicio de un listado.

```
<div class="errores">
  <ul>
    <!--Aca irán los errores--&gt;
  &lt;/ul&gt;
&lt;/div&gt;</pre>
```

Mostrando errores con Javascript

Por último, en nuestro archivo JavaScript, recorreremos con un `for`, el array de errores para que sean mostrados dentro del `<div> `.

```
let ulErrores = document.querySelector("div.errores ul");
for (let i = 0; i < errores.length; i++) {
  ulErrores.innerHTML += "<li>" + errores[i] + "</li>";
}
```

Object Location

“

Javascript nos permite acceder a todos los elementos de nuestra ventana incluyendo nuestra URL. Para poder trabajar con ella, usamos **Object Location**.



Location

Acceder a él es tan sencillo como escribir **Location** en nuestro archivo Javascript. Con él, vienen una serie de métodos y atributos muy interesantes.

- Si usamos el atributo **href** nos devuelve toda la url.

```
{ } console.log(location.href);
```

- Si queremos recargar la página, podríamos usar el método **reload**.

```
{ } location.reload();
```

Query String

Cuando realizamos una petición por **GET**, comúnmente una búsqueda, **Location** nos provee un atributo llamado **search**, para obtener el **query string** entero.



```
{ } console.log(location.href);
```

UrlSearchParams

La interfaz **URLSearchParams** define métodos útiles para trabajar con los parámetros de búsqueda de una URL.

```
{ } let query = new URLSearchParams(location.search);
```

Si queremos preguntar si nuestra **Query String** tiene cierto **parámetro**, usamos el método **has**.

```
{ } query.has('search_query');
```

Si queremos saber que valor tiene ese **parámetro** usamos el método **get**.

```
{ } query.get('search_query');
```

3

4

5



Introducción a API's

APIs Públicas

Las públicas suelen ser APIs que no necesitan registración, ni ningún tipo de solicitud para usarlas. Un ejemplo de estos casos es **RESTCountries**, que nos devuelve información sobre todos los países del mundo.

```
[{
  "Name": "Argentina",
  "topLevelDomain": [".ar"], ...}
```

APIs Semi-Públicas

Por otro lado tenemos el caso de **GIPHY** o **Marvel** que necesitamos **registrarnos** en sus páginas para poder acceder a ellas. Por esto es que las llamamos semi-públicas.

 <https://developers.giphy.com/>
 <https://developer.marvel.com/>

APIs Privadas

Y por último, existen casos como el de **Netflix** que usan APIs internas para poder comunicarse entre diferentes aplicaciones pero que son **privadas**. Es decir, los **endpoints** no están disponibles para nuestro consumo.

 NETFLIX

Endpoints

Cuando hablamos de APIs, **endpoint** hace referencia al punto de conexión donde necesitamos **apuntar** para obtener la información que queremos.

Son las **URLs** que debemos utilizar para obtener la data de un servidor a través de una **API**

- **Ministerio de Modernización**
<https://datos.gob.ar/dataset>
- **Listado de países**
<https://restcountries.eu/rest/v2/>

AJAX Fetch

fetch()

Fetch recibe como primer parámetro la URL del **endpoint** al cual estamos haciendo el llamado asíncrono. Al no saber cuando se completa la petición, el servidor devuelve una **promesa**.

```
{ } fetch("https://restcountries.eu/rest/v2/")
```

1er then

El primer **then** será el encargado de recibir un **callback** y retornará la respuesta de ese llamado **asíncrono** en formato JSON.

```
{ } fetch("https://restcountries.eu/rest/v2/")
  .then(function(response){
    return response.json();
  })
```

2do then

El segundo **then** será el encargado de recibir un **callback** el cual hará lo que le pidamos con la respuesta obtenida en formato JSON.

```
{ } fetch("https://restcountries.eu/rest/v2/")
  .then(function(response){
    return response.json();
  })
  .then(function(dataDecode){
    console.log(dataDecode);
  })
```

3

2

...Y si no funciona el servidor?

En el caso de haber algún error, el **catch()** se encargará de atraparlo y luego lo imprimirá por consola.

```
{ } fetch("https://restcountries.eu/rest/v2/")
  .then(function(response){
    return response.json();
  })
  .then(function(dataDecode){
    console.log(dataDecode);
  })
  .catch(function(error){
    console.log(error);
  });
})
```

Fetch por POST

Como enviaremos datos por **POST**, debemos configurar un **objeto literal** con los datos necesarios para que la API entienda nuestra **petición**, para ello, fetch posee un parámetro opcional.

```
{ } fetch(url,{
  method: 'POST',
  body: JSON.stringify(data),
  headers: {
    'Content-Type': 'application/json'
  }
})
```

5

6

Local Storage y Session Storage

INTRODUCCIÓN

La función de ambos es **almacenar datos** en el navegador.

- **LocalStorage:** permite almacenarlos por tiempo indeterminado
- **SessionStorage:** los guarda en sesión. Es decir, si usamos esta última opción y cerramos el navegador, perderemos todos los datos.

setItem y getItem

El método **setItem** nos permite agregar un nuevo atributo a nuestro **sessionStorage**, en este caso le pusimos "Tom" al atributo "miGato".

Luego, para obtener ese atributo, usaremos el método **getItem**.

```
// Almacena la información en sessionStorage
sessionStorage.setItem('miGato', 'Tom');

// Obtiene la información almacenada desde sessionStorage

let data = sessionStorage.getItem('miGato');
```

removeItem y clearItem

En el caso que queramos **borrar** un ítem en específico, podremos usar el método **removeItem**.

Así mismo, si queremos **borrar todos** los ítems, usaremos el método **clear**

```
// Borra el ítem 'miGato' en sessionStorage
sessionStorage.removeItem('miGato');

// Borra todos los items de sessionStorage

let data = sessionStorage.clear();
```

“

HTTP

HTTP es un **protocolo de transferencia** que permite de manera estandarizada la comunicación entre el **cliente** y el **servidor**.



“

REST

REST es un tipo de arquitectura de servicios que **proporciona estándares** entre sistemas informáticos para establecer **cómo** se van a comunicar entre sí.



REST: CLIENTE - SERVIDOR

CLIENTE - SERVIDOR

Uno de estos estándares es el de **cliente - servidor** el cual indica que la aplicación del cliente y la aplicación del servidor deben poder desarrollarse, extenderse, cambiarse, sin interferir una con otra.

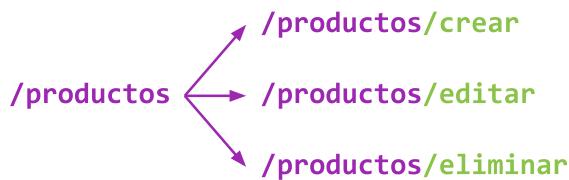
El cliente nunca debería enterarse que se cambió algo del servidor y el servidor no debería tener ningún inconveniente con cambios que realice el cliente.



Lo único que el cliente debería conocer del servicio son los accesos a los recursos (endpoints o URIs)

RECURSOS UNIFORMES

Un recurso en el sistema debería tener solamente un identificador lógico, y éste proveer acceso a toda la información relacionada.



7

STATELESS

El paradigma REST propone que todas las interacciones entre Cliente y Servidor deben ser tratadas como nuevas y de forma absolutamente independiente.

Por lo tanto, si quisieramos tener una aplicación que distinga de usuarios logueados o invitados, debemos mandar toda la información de autenticación necesaria en cada petición.



Esto permite desarrollar aplicaciones más confiables, performantes y escalables, ya que permite que puedan ser dirigidas y actualizadas incluso mientras estas se encuentran funcionando.

8

CACHEABLE

En el paradigma de REST el **cacheo de datos** es una herramienta muy importante, que se implementa del lado del cliente, para mejorar la performance y reducir la demanda al servidor.

El Caché debe aplicarse a los recursos tanto como sea posible.

"

El **cacheo de datos** se utiliza para guardar información que no suele cambiar usualmente, en el equipo del **cliente**. Esto mejora el rendimiento y reduce las peticiones al **servidor**.



9

FORMATOS DE ENVÍO DE DATOS

JSON

Es el formato más común para el envío de datos.

Cuando queramos enviar datos en formato JSON debemos agregar un encabezado en los headers que diga:

`"Content-Type": "application/json"`

12

RAW

Se utiliza para mandar datos con texto sin ningún formato en particular.

No suele ser tan utilizado.

TEXT

Se utiliza para enviar datos que no sean en formato JSON como archivos html, css.

13

URL-encoded

Indica que se nos van a enviar datos codificados en forma de URL. Por lo tanto, nos envía algo muy similar a un query string.

Un dato enviado mediante este método se vería de la siguiente manera:

```
email%3Dcosme%40ulanito.fox  
%26password%3Dverysecret
```

14



MÉTODO HEAD

HEAD

Es un método HTTP que permite conocer la **fecha de la última modificación** de un recurso.

Se utiliza en conjunto con el método GET, de tal manera que si la fecha de modificación no coincide con la del recurso que el cliente tiene cacheada localmente se puedan pedir los datos para actualizarlos.

16

"

Existen más métodos para diseñar una arquitectura o api rest.

Podés investigarlos escaneando el código qr!



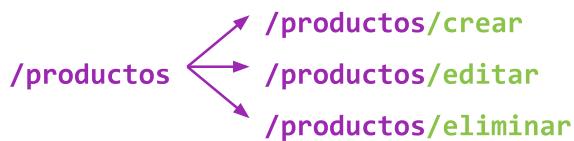
CREACIÓN DE API PROPIA

1.

IDENTIFICAR EL OBJETO MODELO

¿CÓMO IDENTIFICAMOS EL OBJETO MODELO?

El objetivo es identificar los objetos que serán presentados como recursos. La idea es que encontremos un sustantivo relacionado a nuestro negocio, para proveer acceso mediante este a la mayor cantidad de espacios de nuestro sitio. Por ejemplo: “**productos**”



“

Es una buena práctica utilizar siempre sustantivos y encontrar uno que pueda englobar la mayor cantidad de pedidos a nuestro sitio.



3

2.

CREAR LOS URLs

“

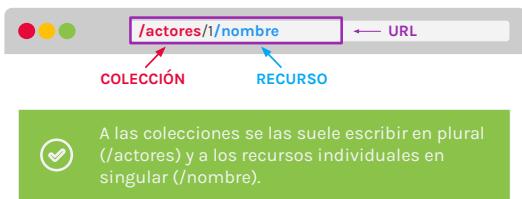
Los URLs o **endpoints** son los identificadores de los recursos de nuestro sitio. Y mediante ellos es como accedemos a los mismos.



COLECCIONES

Las **colecciones** en REST son un recurso por el cual puedo acceder a un conjunto de recursos o **elementos**.

A cada elemento por separado se lo llama **recurso**.



¿QUÉ NOS DEVUELVEN LOS URLs?

Cuando accedemos a una URL de nuestra API, esta nos suele devolver datos en formato **JSON**.

Algunos de esos datos son:

- Una clave link que apunta al mismo endpoint al que acabamos de acceder.
- Información genérica (ej. Cantidad de datos)
- Datos básicos de la petición y otros endpoints para acceder a más detalles.

7

8

2.

OPERACIONES DE LAS RUTAS

OPERACIONES DE LAS RUTAS

Existen 5 operaciones asociadas a las rutas:

- **GET**: Permite acceder a todos los datos del recurso.
- **POST**: Permite crear un nuevo recurso.
- **PUT**: Reemplaza un recurso ya existente.
- **PATCH**: Actualiza o modifica parcialmente un recurso ya existente.
- **DELETE**: Permite indicar al servidor que borre un recurso.

10

“

La finalidad de **Axios** es hacer comunicaciones HTTP a través de código, **configurando solicitudes como objetos de Javascript**.



AXIOS

¿CÓMO APLICAR AXIOS?

INCORPORAR LA LIBRERÍA

Incorporando la librería al backend:

Para incorporar la librería al backend simplemente tenemos que ejecutar en nuestra consola el comando:

```
0 npm install axios
```

Incorporando la librería al front-end:

Para incorporar la librería al front-end podemos incluir una etiqueta con el vínculo correspondiente o utilizar un CDN, que se vería de la siguiente forma:

```
<script  
src="https://cdnjs.cloudflare.com/ajax/libs/axios/0.1  
9.1/axios.min.js"></script>
```

ARMAR LAS PETICIONES

Las peticiones tienen dos lados: el de **creación** y el de **consumo**. Comenzaremos armando la parte de creación:

- Primero crearemos una carpeta dentro de `/src` que se llame **requests** de manera que podamos tener todas las peticiones de cada recurso dentro del mismo directorio.
- Luego crearemos dentro de la carpeta **requests**, el archivo **default.js** este archivo genera un objeto que va a representar todos los parámetros por default que vamos a utilizar en cada pedido.

DEFAULT.JS

```
const default = {  
  baseURL: 'www.spotify.com/api/',  
  timeout: 4000  
};  
  
module.exports = default;
```

Archivo default.js

timeout: representa el **tiempo máximo** de espera de respuesta para una solicitud. Si el servidor demora más tiempo, **Axios** cancelará la solicitud.

baseURL: su valor es la url que va a usarse en **todos** los pedidos que hagamos.

ARMAR UN REQUEST

- Primero identificamos a **qué recurso** queremos acceder y así poder crear un archivo, que va a contener nuestro pedido y todos los que interactúen con este recurso.

Dentro de este archivo tendremos dos partes:

- La primera contendrá la **url que identifica al recurso** y además la importación de Axios y de los defaults.
- La segunda será un objeto literal que representará los servicios que serán funciones a las cuales podremos acceder por el nombre de la clave.

GET REQUEST

```
const axios = require('axios');
const default = require('./default');
const url = 'artista';
const artistRequest = {
  getArtista: function(id){
    return axios({
      ...defaults,
      method: 'get',
      url: `${url}/${id}`
    });
  }
};
module.exports = artistRequest;
```

get request

7

8

CONSUMIR UN REQUEST

- En donde necesitemos usar el servicio, primero importamos el módulo y lo almacenamos en una variable para invocar los métodos que contenga.

```
0 const artistRequest = require('./requests/src/artistsRequest');

1 artistRequest.getArtista(59).then(response => {
2   console.log(response.data);
3 }).catch(error => {
4   console.log(error.response);
5 });
```

9

“

POSTMAN

Postman es una herramienta que nos simplifica mucho nuestras tareas a la hora de testear el funcionamiento de una **API** sin tener el front end o scripts hechos.



POSTMAN: INSTALACIÓN

INSTALACIÓN

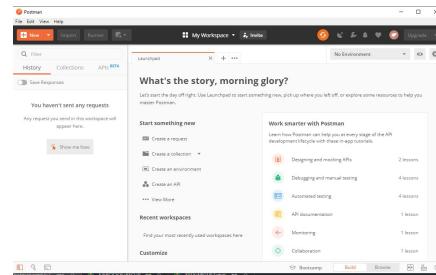
Lo primero que haremos será visitar el sitio oficial de Postman <https://www.getpostman.com/> y descargar el instalable que corresponda al sistema operativo que usamos. Luego, por supuesto, debemos instalarlo.



POSTMAN: INTERFAZ

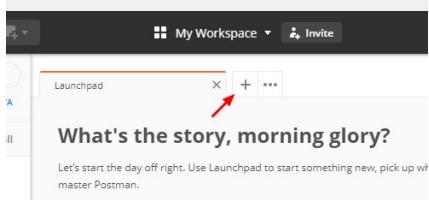
INTERFAZ

En el panel de la izquierda encontraremos, entre otras opciones, el historial de todas las peticiones que hayamos realizado. Obviamente podremos borrar alguna o todas. En el panel de la derecha es donde más vamos a operar.



INTERFAZ

Ahora bien, haremos nuestra primera petición por **GET** a un **endpoint** previamente visto. Para ello haremos click en el símbolo +.



INTERFAZ

Luego, ingresamos nuestra URL a probar. En este caso elegimos <https://restcountries.eu/rest/v2/> para listarnos todos los países.

7

8

INTERFAZ

Finalmente haremos click en **SEND** para ver los resultados. Como podemos observar, hay un indicador de estado a la derecha, otro indicador de cuánto tiempo demandó realizar la petición, el tamaño del Json y debajo se encuentra el resultado.

```
1 [ {  
2   "name": "Afghanistan",  
3   "topLevelDomain": [  
4     ".af"  
5   ],  
6   "alpha2Code": "AF",  
7   "alpha3Code": "AFG",  
8   "callingCodes": [  
9     "+93"  
10  ],  
11  "capital": "Kabul",  
12  "altSpellings": [  
13    "AF",  
14  ]  
15 } ]
```

TESTS

Para comenzar a escribir **tests** nos dirigimos a la solapa "Tests". En el espacio en blanco podemos escribir utilizando la sintaxis de Javascript que ya conocemos.

Para ello usaremos las funciones de PostMan "**pm**" que nos permiten realizar los tests.

```
pm.test('salio todo ok!', function () {  
  pm.response.to.have.status(200)  
})  
  
pm.test('la respuesta es json', function () {  
  pm.response.to.be.json  
})  
  
pm.test('salio todo ok!', function () {  
  pm.response.to.have.status(200)  
})
```

9

10

React Instalación

React Instalación

Donde queramos crear nuestro proyecto en React, debemos ejecutar los siguientes comandos.

```
>_ npm init react-app 'mi-primer-proyecto'  
>_ cd 'mi-primer-proyecto'  
>_ npm start
```



3

React Instalación

Al ser una librería de Javascript, React pertenece al conjunto de paquetes que gestiona NPM. Es decir, necesitamos NPM para instalar el ambiente.



2

React Instalación

```
>_ cd 'mi-primer-proyecto'
```

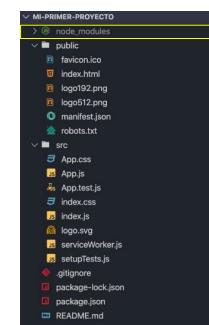


Ingresamos a la raíz de nuestro proyecto.



5

React Estructura de carpetas



En **node_modules** están todos los paquetes de **NPM** instalados para que nuestro proyecto en React funcione correctamente.



6

React

Estructura de carpetas

```
Mi-PRIMER-PROYECTO
> node_modules
  - public
    - favicon.ico
    - index.html
    - logo192.png
    - logo512.png
    - manifest.json
    - robots.txt
  - src
    - App.css
    - App.js
    - App.test.js
    - index.css
    - index.js
    - logo.svg
    - serviceWorker.js
    - setupTests.js
    - .gitignore
    - package-lock.json
    - package.json
    - README.md
```

En la carpeta **public** se encuentra el archivo **index.html**. Este es el archivo html principal que se va cargar cuando el usuario ingresa a la **URL** de nuestra aplicación.

Dentro de este encontraremos una etiqueta **div** con **id "root"** donde se insertará nuestra aplicación React.



7

React

Estructura de carpetas

```
Mi-PRIMER-PROYECTO
> node_modules
  - public
    - favicon.ico
    - index.html
    - logo192.png
    - logo512.png
    - manifest.json
    - robots.txt
  - src
    - App.css
    - App.js
    - App.test.js
    - index.css
    - index.js
    - logo.svg
    - serviceWorker.js
    - setupTests.js
    - .gitignore
    - package-lock.json
    - package.json
    - README.md
```

En la carpeta **src** están todos los archivos donde vamos a trabajar en nuestro proyecto en React.

También encontrarás un archivo llamado **index.js**, encargado de renderizar nuestro div con **id "root"**.



8

React

Estructura de carpetas

```
Mi-PRIMER-PROYECTO
> node_modules
  - public
    - favicon.ico
    - index.html
    - logo192.png
    - logo512.png
    - manifest.json
    - robots.txt
  - src
    - App.css
    - App.js
    - App.test.js
    - index.css
    - index.js
    - logo.svg
    - serviceWorker.js
    - setupTests.js
    - .gitignore
    - package-lock.json
    - package.json
    - README.md
```

En **package.json** se especifican las dependencias y las versiones de los paquetes de los que depende el proyecto.



9

React

Instalación

```
>_ npm start
```



Una vez estemos en la raíz de nuestro proyecto nos queda solamente iniciarla. Este comando nos permite iniciar el servidor. Cuando lo ejecutemos se nos abrirá una página con la aplicación de React por defecto que viene con la instalación.



10

React

Instalación

Si en el navegador, en la URL '**localhost:3000**' se ve algo así, se instaló React correctamente.



React

Documentación

Para saber más puedes acceder a la documentación oficial de ReactJS haciendo click en el siguiente [Link](#):



11

12

React Ecosistema

Llamamos **ecosistema React** al conjunto de herramientas adicionales, aplicaciones y librerías que permite a **react.js** trabajar como un framework.



Ecosistema Una cadena de herramientas

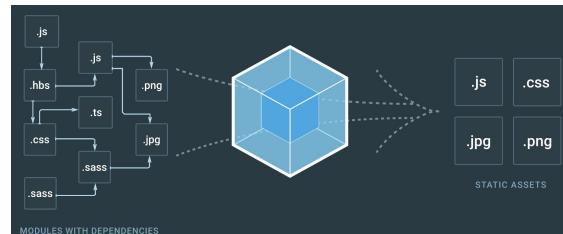
Webpack permite escribir código modular y empaquetarlo para optimizar el tiempo de carga.

Babel es un compilador. Permite escribir Javascript moderno (**ES6**) y que aún así funcione en navegadores más antiguos. **Babel** toma el código nuevo y lo traduce a la última versión estable aceptada por los navegadores.

3

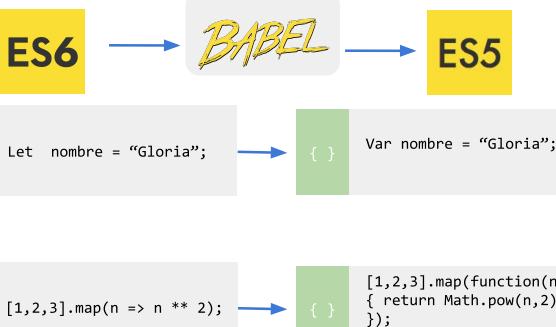
Webpack

Webpack se puede considerar como un Task Runner muy especializado en el procesamiento de unos archivos de entrada para convertirlos en otros archivos de salida, para lo cual utiliza unos componentes que se denominan **loaders**.



4

Babel



5

Ecosistema Una cadena de herramientas

Para saber más puedes acceder a la documentación oficial de React, haciendo click en el siguiente [Link](#):



React Introducción a componentes

Los **componentes** permiten separar la interfaz de usuario en **piezas independientes reutilizables** pensando cada pieza de forma aislada.
Técnicamente son **funciones** nativas de **JavaScript**.



REACT

Componentes stateless

Se llaman componentes **stateless** (componentes funcionales o componentes sin estado) porque internamente implementan un **retorno con lógica sencilla** que entrega una estructura **html**.

Dentro de la lógica sencilla eventualmente podemos encontrar estructuras de control (“if”) y seguramente varios métodos map para iterar arrays.

```
function Navbar() {
  return (
    <nav>
      <a href="/home">Home</a>
      <a href="/productos">Productos</a>
    </nav>
  );
}
```

Los componentes ÚNICAMENTE pueden devolver 1 sólo elemento html.

Dentro del **return** no puede haber etiquetas “hermanas”.

3

REACT

Crear un componente statless

Crea un archivo con extensión .js con el nombre de la función que llevará dentro. En este caso: /src/components/Navbar.js

```
{ } import React from "react";

{ } function Navbar() {
  return (
    <nav>
      <a href="/home">Home</a>
      <a href="/productos">Productos</a>
    </nav>
  );
}

{ } export default Navbar;
```

REACT

Implementar un componente stateless

En el archivo App.js importamos el componente Navbar.

Luego lo implementamos como si fuese una etiqueta de HTML dentro de la estructura del **return** del componente App.

No olvides poner una barra (/) al final para cerrar la etiqueta.

```
{ } import Navbar from "./components/Navbar";

{ } function App() {
  return (
    <div className="App">
      <Navbar />
    </div>
  );
}
```

5

REACT

Introducción a componentes

Para saber más sobre componentes puedes acceder a la documentación oficial de React haciendo click en el siguiente [Link](#):



React JSX

“

JSX es una extensión de la sintaxis de JavaScript.

Se usa para describir cómo debería ser la interfaz de usuario.

JSX puede recordarte a un lenguaje de plantillas pero viene con todo el poder de JavaScript.



REACT Insertando expresiones

En el ejemplo a continuación, declaramos una variable llamada `name` y luego la usamos dentro del JSX envolviéndola dentro de llaves:

```
const name = 'John Doe';
const element = <h1>Hello, {name}</h1>

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

3

REACT Insertando expresiones

Se puede colocar cualquier expresión de JavaScript dentro de llaves en JSX. Por ejemplo, `2 + 2`, `user.firstName`, o `formatName(user)` son todas expresiones válidas de Javascript.

En el ejemplo a continuación, insertamos el resultado de llamar la función de JavaScript, `formatName(user)`, dentro de un elemento `<h1>`.

REACT Insertando expresiones

```
const user = {
  firstName: 'John',
  lastName: 'Doe',
  formatName: function () {
    return this.firstName + ' ' + this.lastName;
  }
}

const element = (
  <h1>
    Hello, {user.formatName()}!
  </h1>
);

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

5

¡ATENCIÓN !

Dado que JSX es más cercano a JavaScript que a HTML, React DOM usa la convención de nomenclatura camelCase en vez de nombres de atributos HTML. Por ejemplo, `class` se vuelve `className` en JSX, y `tabindex` se vuelve `tabIndex`.



```
const element = (
  <h1 className="greeting">
    Hola, Mundo!!!
  </h1>
);
```

2

4

6

¡ ATENCIÓN !



Si una etiqueta está vacía, puedes cerrarla inmediatamente con />

```
{ }
```

```
const element = <img src={user.avatarUrl} />;
```



REACT JSX

Para saber más puedes acceder a la documentación oficial de Sequelize haciendo click en el siguiente [Link](#)

7



8

Desarrollo Web Full Stack Node

GUÍA Práctica 1 de React

Para la práctica de estos puntos en React consideramos que lo mejor es tener la mayor experiencia real posible, por lo cual vamos a estar trabajando con un proyecto localmente.

Abajo vas a encontrar una serie de ejercitaciones y luego la solución de cada una de las mismas. Te recordamos que la solución está para ayudarte en el caso de que la necesites. Si logras resolverlo con la solución te invitamos a que la borres y luego vuelvas a plantear la misma sin consultar la solución directamente así comprobamos que adquiriste y desarrollaste los conceptos.

Te mostramos una imagen final de lo que vas a estar realizando enteramente en REACT.



Mi primer sitio con React JS



Fuente: <https://placeimg.com/500/240/nature>

El texto de relleno es simplemente el texto de relleno de las imprentas y archivos de texto. El texto de relleno ha sido el texto de relleno estándar de las industrias desde el año 1500, cuando un impresor (N. del T. persona que se dedica a la imprenta) desconocido usó una galería de textos y los mezcló de tal manera que logró hacer un libro de textos especímen. No sólo sobrevivió 500 años, sino que también ingresó como texto de relleno en documentos electrónicos, quedando esencialmente igual al original. Fue popularizado en los 60s con la creación de las hojas "Letraset", las cuales contenían pasajes de Lorem Ipsum, y más recientemente con software de autoedición, como por ejemplo Aldus PageMaker, el cual incluye versiones de Lorem Ipsum. Al contrario del pensamiento popular, el texto de Lorem Ipsum no es simplemente texto aleatorio. Tiene sus raíces en una pieza clásica de la literatura del Latín, que data del año 45 antes de Cristo, haciendo que este adquiera más de 2000 años de antigüedad. Richard McClintock, un profesor de Latín de la Universidad de Hampden-Sydney en Virginia, encontró una de las palabras más oscuras de la lengua del latín, "consecetur", en un pasaje de Lorem Ipsum, y al seguir leyendo distintos textos del latín, descubrió la fuente indudable. Lorem Ipsum viene de las secciones 1.10.32 y 1.10.33 de "de Finibus Bonorum et Malorum" (Los Extremos del Bien y El Mal) por Cicero, escrito en el año 45 antes de Cristo. Este libro es un tratado de teoría de éticas, muy popular durante el Renacimiento.

El texto de relleno es simplemente el texto de relleno de las imprentas y archivos de texto. El texto de relleno ha sido el texto de relleno estándar de las industrias desde el año 1500, cuando un impresor (N. del T. persona que se dedica a la imprenta) desconocido usó una galería de textos y los mezcló de tal manera que logró hacer un libro de textos especímen. No sólo sobrevivió 500 años, sino que también ingresó como texto de relleno en documentos electrónicos, quedando esencialmente igual al original. Fue popularizado en los 60s con la creación de las hojas "Letraset", las cuales contenían pasajes de Lorem Ipsum, y más recientemente con software de autoedición, como por ejemplo Aldus PageMaker, el cual incluye versiones de Lorem Ipsum. Al contrario del pensamiento popular, el texto de Lorem Ipsum no es simplemente texto aleatorio. Tiene sus raíces en una pieza clásica de la literatura del Latín, que data del año 45 antes de Cristo, haciendo que este adquiera más de 2000 años de antigüedad. Richard McClintock, un profesor de Latín de la Universidad de Hampden-Sydney en Virginia, encontró una de las palabras más oscuras de la lengua del latín, "consecetur", en un pasaje de Lorem Ipsum, y al seguir leyendo distintos textos del latín, descubrió la fuente indudable. Lorem Ipsum viene de las secciones 1.10.32 y 1.10.33 de "de Finibus Bonorum et Malorum" (Los Extremos del Bien y El Mal) por Cicero.

También te dejamos el [código resuelto](#) como guía y ayuda por si no puedes avanzar.

Preparación de ambiente

Para comenzar con React tenemos que analizar si tenemos todas las herramientas necesarias y no hay mejor forma que intentar crear un proyecto con React ejecutando los siguientes comandos:

Instalación de proyecto base:

```
>npm init react-app 'mi-primer-proyecto'
```

Ejecución del proyecto:

```
>npm start
```

Estos comandos deberían de haberse ejecutado exitosamente, aunque si durante la ejecución hubo algún inconveniente te pedimos que lo comuniques por slack para poder darte soporte.

Ejercitación

PRACTICA 1: Arreglando el proyecto

El siguiente [proyecto](#) que vas a bajar no puede renderizar bien la pagina. Te recordamos que al bajarlo debes correr el comando [npm install](#) para instalar las dependencias y luego [npm start](#) para ejecutar el proyecto.

¿Se te ocurre el porqué falla?--> Trata de analizar 5 minutos como se levanta el proyecto de React, cuales son los primeros componentes en levantarse y observar si eso se cumple en el proyecto, ¿Que hace el componente 'Website.js'?

A practicar :).

TIP: React tiene un componente llamado ReactDOM que es el encargado de renderizar los mismos, ¿Se está renderizando el componente correcto?.

PRACTICA 2: Creá tu primer componente

¡Genial! Ahora el componente Website devuelve un hermoso 'Hola Mundo'.

Este es el primer componente que está renderizando TODO el sitio web. Por lo cual sería una buena práctica modularizar TODO el sitio en varios componentes, pero antes de comenzar con eso puntualmente trataremos de crear el primer componente nuestro desde 0.

Tu tarea va a ser mover el "Hola Mundo" a un componente llamado HelloWorld.js. Luego el mismo componente tiene que ser importado y usado en el componente Website.

¡Vos podes!

TIP: Te recordamos que el código para crear un componente es el siguiente:

```
function NombreComponente () {  
  return (  
    <h1>Wendis</h1>  
  );  
  
}  
  
export default NombreComponente;
```



TIP: Te recordamos que el código para poder utilizar un componente es el siguiente:

```
import React from 'react';
import ComponenteHijo from './components/Componente
function ComponentePadre () {
  return (
    <ComponenteHijo/>
  );
}
export default ComponentePadre;
```

PRÁCTICA 3: Agregando un NavBar

Lograste crear tu primer componente y utilizarlo :).

Ahora vamos a agregar un NavBar y SOLO un Navbar en el proyecto. Para esto tenes que [descargar](#) el componente NavBar en el siguiente enlace. El mismo va conformar la barra en la parte superior de nuestro sitio. Tu tarea va a ser sacar el componente HelloWorld y reemplazarlo por el NavBar.

¿Donde va agrego el archivo NavBar.js?

Dentro de la carpeta components.

¿Solo eso hay que hacer?

Casi, son varias cosas.

- 1) Incorpora el componente al proyecto y fijate que el mismo renderice.
- 2) Analizar el componente NavBar en detalle.

Analizando el componente NavBar vas a encontrar que la finalidad del mismo es mostrar en la barra de navegación el avatar y nombre del usuario.

Vas a tener que agregar dos variables, una para el nombre del usuario y otra para el avatar. Esto tenes que hacerlo dentro del archivo del componente pero antes de la definición del mismo para tener **harcodado** la url de la imagen del avatar y el nombre de usuario. Luego vas a tener que analizar dónde y cómo dentro del componente agregar esas variables para poder renderizarlo, hay algunas pistas,

recorda que para renderizar JS dentro vas a tener que escribir el mismo dentro de las llaves "{}".

Son varias cosas por lo cual es normal que te lleve un poco más de tiempo.

TIP: Te recordamos que el código para agregar variables al componente y utilizarlo es similar al siguiente:

```
import React from 'react'

let mensajeAMostrar = 'Wendis'

function Componente() {
    return (
        <h1>{mensajeAMostrar}</h1>
    )
}

export default Componente
```

PRÁCTICA 4: Más componentes

Lograste incluir el NavBar, ¡muy bien!.

Ahora vamos a agregar los siguientes componentes:

[Content](#)-> Tendrá el contenido principal de nuestro sitio web.

[Footer](#)-> Será el pie de nuestro sitio web

[FooterLinks](#)-> Este componente va a representar un conjunto de enlaces que van a aparecer dentro del footer.

TIP: Recorda que cada componente de React debe devolver un componente HTML por lo cual te sugerimos que WebSite devuelva todos los componentes dentro de la siguientes estructura:

```
<div className="container-fluid">
    {/* Aca van los componentes */}
```

```
</div>
```

¿Ya los agregue, hay que hacer algo más?

Si, cada uno de estos componentes también van a requerir que agregues algo en ellos en forma similar a lo que hiciste con NavBar.

Content

```
título-> "Mi primer sitio con React JS"  
parrafo-> require("../data/articulo.js").texto;  
image -> "https://placeimg.com/500/240/nature"
```

Footer

El componente debe contener 3 FooterLinks.

FooterLinks

Solo debes agregar este componente dentro del componente Footer.

Ejercitación Soluciones:

PRACTICA 1: Arreglando el proyecto:

Dentro del archivo index.js:

- 1) Renderiza el componente a través de ReactDOM

```
ReactDOM.render(<Website />, document.getElementById('root'));
```

PRACTICA 2: Creá tu primer componente

- 1) Crear el archivo HelloWorld.js

```
import React from 'react';

function HelloWorld() {
  return (
    <h1>Hola Mundo</h1>
  );
}
```

```
export default Website;
```

- 2) Importarlo en Website.js y renderizarlo

```
import React from 'react';

import HelloWorld from './components/HelloWorld'

function Website() {
  return (
    <HelloWorld/>
  );
}

export default Website;
```

PRÁCTICA 3: Agregando un NavBar

- 1) Agregar el archivo Navbar.js
- 2) Importarlo y usarlo en Website.js

```
import React from 'react';
import Navbar from './components/Navbar'

function Website() {
  return (
    <Navbar/>
  );
}

export default Website;
```

- 3) Dentro de Navbar.js agregar las dos variables y utilizarlas.

```
import React from 'react'
let username = 'Alejandro'
let avatar = 'https://i.pravatar.cc/150?img=39'

function Navbar() {
  return (
    <nav class="navbar navbar-light bg-light">
      <a class="navbar-brand" href="#">
        <img
          src={avatar} width="30" height="30"
          class="d-inline-block align-top rounded-circle
          mr-3" />
        {username}
      </a>
    </nav>
  )
}
```

```
export default Navbar
```

PRÁCTICA 4: Más componentes

1) Tenes que agregar los archivos al proyecto.

2) Importarlos y usarlos en Website.js

```
import React from 'react';
import Navbar from './components/Navbar'
import Content from './components/Content'
import Footer from './components/Footer'

function Website() {
  return (
    <div className="container-fluid">
      <Navbar/>
      <Content/>
      <Footer/>
    </div>
  );
}

export default Website;
```

3) Completar las funcionalidades de los componentes

Content

título-> "Mi primer sitio con React JS"
 parrafo-> require("../data/articulo.js").texto;
 image -> "<https://placeimg.com/500/240/nature>"

```
import React from 'react'

let titulo = 'Mi primer sitio con React JS';
```

```
let parrafo = require('../data/articulo.js').texto;

let image = 'https://placeimg.com/500/240/nature';

function Content() {

    return (

        <div className="row my-3">

            <div className="col">

                <div className="row">

                    <div className="col">

                        <h1>{titulo}</h1>

                    </div>

                </div>

            </div>

        <div className="row my-3">

            <div className="col-md-4">

                <img style={{width:'100%'}}>

                src={image} alt={'articulo'}/>

                <a target="_blank"
                className="text-secondary" href={image}>Fuente:
                {image}</a>

            </div>

        </div>

        <div className="col-md-8">
```

13

```
<p>{parrafo}</p>

</div>

</div>

<div className="row my-3">

  <div className="col">

    <p>{parrafo}</p>

    </div>

  </div>

</div>

</div>

)
```

```
export default Content
```

Footer

```
import React from 'react'

import FooterLinks from './FooterLinks'

function Footer() {

  return (
    <footer>
```

14

```
        <div class="row text-center text-xs-center text-sm-left text-md-left bg-light">  
            <FooterLinks />  
            <FooterLinks />  
            <FooterLinks />  
        </div>  
    </footer>  
)  
}  
  
export default Footer
```

FooterLinks

Solamente hay que incorporarlo al proyecto

React Props

Las props son entradas de un componente de React. Representan información que es pasada desde un componente padre a un componente hijo, pueden recibir propiedades como parámetros para poder insertar valores y eventos en el HTML.



REACT

Definiendo una props

Esta función es un componente de React válido porque acepta un solo argumento de objeto “props” (que proviene de propiedades) con datos y devuelve un elemento de React. Llamamos a dichos componentes “funcionales” porque literalmente son funciones JavaScript.

```
function Bienvenido(props) {  
  return (  
    <h1>  
      Hola, { props.nombre }  
    </h1>  
  );  
}
```

La forma más sencilla de definir una props es escribir una función de JavaScript.

El componente no retornará texto fijo, sino que el mismo retornará el contenido de acuerdo al valor de sus propiedades.

3

REACT

Pasando props al componente

Las propiedades de un componente reciben sus respectivos valores, cuando el componente es invocado por la aplicación

```
<Bienvenido  
  nombre = 'Victor Luis' />
```

React trata los componentes que empiezan con letras minúsculas como etiquetas del DOM. Por ejemplo, `<div>` representa una etiqueta div HTML pero `<Bienvenido />` representa un componente y requiere que el mismo esté definido.

4

React Documentación

Para saber más puedes acceder a la documentación oficial de ReactJS haciendo click en el siguiente [Link](#):



React Key Props y map

REACT Recibir las props en el componente

Dentro del **componente**, lo primero que tenés que hacer es recibir las **props** como parámetro de la función. Luego, dentro de la estructura de JSX que hayas definido, vas a tener que iterar sobre el array recibido para poder imprimir los usuarios.

```
function MiLista(props) {
  return (
    <ol> {items.map(item => <li> {item} </li>)}
    </ol> );
}
```

El método que conviene implementar para iterar sobre el array e imprimir su contenido en React es el **map()**.

3

REACT Pasando data en propiedades

Observá como el array es pasado dentro del atributo con nombre **items** y que adicionalmente va entre llaves, porque son estas llaves, las que te permiten escribir tipos de datos de javascript que no sean simplemente cadenas de texto.

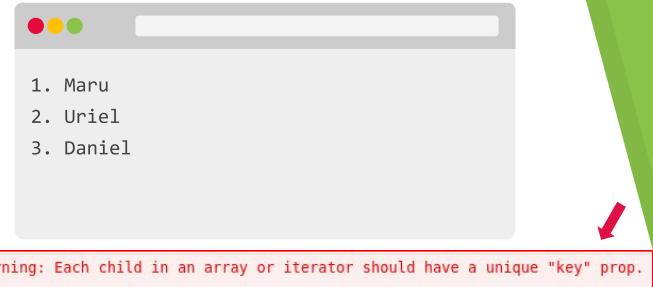
```
const usuarios = ["Maru", "Uriel", "Daniel"];
<MiLista
  items = { usuarios } />
```

En este ejemplo podemos observar que estamos pasando al componente **MiLista**, los usuarios que se encuentran en el array **usuarios** y los mismos son asignados a la variable **items**.

2

REACT Así se ve la vista en el navegador

Debemos estar atentos al error que nos aparece en las Dev Tools.



4

“

Las **key** ayudan a React a identificar qué elementos han cambiado, agregado o eliminado.

Es decir, React por medio de las keys determina si es el mismo elemento o no.

- ▶ Solo es necesario agregar keys cuando devolvemos un array de elementos iguales.
- ▶ La key debe ser única entre elementos hermanos.
- ▶ Las keys no se muestran en el HTML final (si quisieramos esto también deberíamos utilizar id)



5

REACT Key props

Los componentes que renderizan varios elementos del mismo tipo necesitan de una **key** con valor único, porque la misma ayuda a React a identificar qué **ítems** cambiaron, cuáles se agregaron o cuáles se eliminaron.

```
function MiLista(props) {
  return (
    <ol> {items.map(item , i) => <li> {item + i}
      </li>)}
    </ol> );
```

Las **keys** deben ser dadas a los elementos dentro de mapeo del array para darle a los elementos una identidad única y estable.

6

REACT

Key Props y map

Para saber más sobre componentes puedes acceder a la documentación oficial de React haciendo click en el siguiente [Link](#):



React PropTypes y DefaultProps

A medida que tu aplicación crece, podés capturar una gran cantidad de errores con verificación de tipos. React tiene algunas habilidades de verificación de tipos en las props de un componente, para ello se puede asignar la propiedad especial **PropTypes**



React Instalación de las propTypes

Las propTypes pertenecen al conjunto de paquetes que gestiona NPM. Es decir valiéndonos de NPM instalamos las mismas.



3



React Instalación propTypes

Para instalar en nuestro Proyecto React las propTypes, debemos ejecutar el siguiente comando.

```
>_ npm install --save prop-types
```



React Instalación propTypes

```
>_ import PropTypes from 'prop-types'; // ES6
>_ var PropTypes = require('prop-types'); // ES5 with
  npm
```



Una vez que el paquete se haya instalado. Se debe importar dicho paquete allá en aquel componente donde querés implementar las propTypes.

5



REACT Definir propTypes en Componentes

PropTypes exporta un rango de validadores que pueden ser usados para estar seguros que la información recibida sea válida.

```
{ } import PropTypes from 'prop-types';
{ }
Function Saludar(props) {
  render() {
    return (
      <h1>Hola, {props.nombre}</h1>
    );
  }
}
Saludar.propTypes = {
  nombre : PropTypes.string
};
{ } Export default Saludar;
```

6

“

defaultProps puede ser definido como una propiedad en el propio componente, para establecer las props predeterminadas que recibirá el mismo.

Esto se utiliza para props no definidos, pero no para props nulos.



7

REACT

Definiendo las `defaultProps`

En el componente debe Definir una propiedad que se llame `defaultProps` a la misma le asignará un objeto literal como valor. Dentro de este objeto literal, es en donde se hará referencia al nombre de la propiedad y se le asignará el valor por defecto que quieras que la `prop` tenga.

```
function SeteoBoton(props) {  
  // ...  
}  
  
SeteoBoton.defaultProps = {  
  color: 'blue'  
};
```

Si `props.color` no es proporcionado, se establecerá por defecto a 'blue':

8

REACT

PropTypes y DefaultProps

Para saber más sobre las `propTypes` puedes acceder a la documentación oficial de React haciendo click en el siguiente [Link](#):



9

REACT CHILDREN

Los **componentes children** son todos aquellos *elementos* que son pasados **entre** las etiquetas de un **componente padre**.



CHILDREN Configuración

Dentro el objeto literal `props`, existe la propiedad `children`. Haciendo uso de esta propiedad, podremos traer a todos los hijos que definamos dentro del componente padre:

```
{ } function Saludo(props) {  
  return (  
    <div>  
      <h1>{ props.titulo }</h1>  
      <p>{ props.mensaje }</p>  
      { props.children }  
    </div>  
  );  
}
```

De esa forma le estamos aclarando al componente **dónde** debería imprimir a los componentes hijos, **en caso de recibir alguno**.

3

CHILDREN Implementación

Cuando llamamos al componente que creamos, tendremos que utilizar **etiquetas de apertura y cierre** para definir dentro el contenido que haga falta.

```
{ } <main>  
  <Saludo>  
    // definimos el contenido A  
  </Saludo>  
  <Saludo>  
    // definimos el contenido B  
  </Saludo>  
</main>
```

4

¿CUÁNDO USAMOS UN CHILDREN?

Cuando no sabemos exactamente qué contenido puede llegar a haber dentro de un contenedor padre. De esta manera, estamos configurando un componente **flexible** y **reutilizable**.



5

REACT STYLING

React **propone** y **permite** tener **una hoja de estilo por componente**, pudiendo de esta forma **modularizar** aún más nuestro **código** y tener mayor **control** sobre el mismo.



1.

Vincular un archivo css al componente:

```
import './hojaEstiloComponente.css';
```

La ruta hacia la hoja de estilo será **relativa** al componente desde donde la estamos importando.

2.

Asignar una clase a un elemento:

```
<h2 className="nombreProducto"> ...
```

Usamos **className** porque **class** es una **palabra reservada** de Javascript.

3.

Aunque estemos modularizando nuestros estilos **NO repetir nombres de clases** entre hojas de estilos. Esto traería conflictos al momento de renderizar el sitio ya que se empezarían a pisar nuestras clases entre sí.

REACT INTRO STATEFULL COMPONENTS

Los **componentes con estado** son aquellos que pueden cambiar su **contenido interno** a partir de **eventos externos**.



REACT Componentes statefull

Los componentes **statefull** -o componentes con estado- son aquellos que están atentos a los eventos que los rodean y saben reaccionar a los mismos. En función de esos eventos, el componente podrá cambiar su contenido interno.

Estos componentes ya no serán funciones nativas de Javascript si no que los trabajaremos como un nuevo tipo de dato: una **clase**.

```
{}
class Producto extends Component {
}
```

{ código }

```
import React, { Component } from 'react';
class NombreComponente extends Component {
  render () {
    return (
      // código a renderizar ...
    );
  }
}

export default NombreComponente;
```

3

2

{ código }

```
import React, { Component } from 'react';
class NombreComponente extends Component {
  render () {
    return (
      // código a renderizar ...
    );
  }
}

export default NombreComponente;
```

Importamos React, aclarando que queremos traer el objeto Component,

{ código }

```
import React, { Component } from 'react';
class NombreComponente extends Component {
  render () {
    return (
      // código a renderizar ...
    );
  }
}

export default NombreComponente;
```

5

4

Usamos la palabra reservada class para empezar a definir nuestro componente.

{ código }

```
import React, { Component } from 'react';
class NombreComponente extends Component {
  render () {
    return (
      // código a renderizar ...
    );
  }
}

export default NombreComponente;
```

Le asignamos el nombre que queremos al componente...

{ código }

```
import React, { Component } from 'react';
class NombreComponente extends Component {
  render () {
    return (
      // código a renderizar ...
    );
  }
}

export default NombreComponente;
```

Usamos la palabra reservada **extends** seguida de la palabra **Component**, la cual hace referencia al objeto que trajimos cuando importamos React.

7

8

{ código }

```
import React, { Component } from 'react';
class NombreComponente extends Component {
  render () {
    return (
      // código a renderizar ...
    );
  }
}

export default NombreComponente;
```

Para poder renderizar los elementos visuales del componente, usamos el método **render()**.

{ código }

```
import React, { Component } from 'react';
class NombreComponente extends Component {
  render () {
    return (
      // código a renderizar ...
    );
  }
}

export default NombreComponente;
```

Llamamos al método **return** y definimos los elementos que queremos renderizar.

9

10

{ código }

```
import React, { Component } from 'react';
class NombreComponente extends Component {
  render () {
    return (
      // código a renderizar ...
    );
  }
}

export default NombreComponente;
```

Exportamos el componente.

11

REACT STATE SETSTATE

El **setState()** programa una actualización al **objeto state** de un componente. Cuando el **state** cambia, el componente responde volviendo a **renderizar**.



REACT constructor

```
Class contador extends Component{
  constructor(){
    super();
    this.state = {
      valor:1,
    }
  }
}
```

El método **constructor** es necesario para poder definir la estructura de un componente.

REACT constructor

```
Class contador extends Component{
  constructor(){
    super();
    this.state = {
      valor:1,
    }
  }
}
```

La función **super** en el constructor es necesaria en react ya que hereda de su clase padre.

3

4

REACT constructor

```
Class contador extends Component{
  constructor(){
    super();
    this.state = {
      valor:1,
    }
  }
}
```

El constructor es el único lugar donde debes asignar **this.state** directamente. Éste va a ser un objeto literal

REACT props

```
Class contador extends Component{
  constructor(props){
    super(props);
    this.state = {
      valor: props.valor,
    }
  }
}
```

Podemos recibir las **props** en el **constructor**, es buena práctica utilizarlas al llamar al **super**

5

6

REACT setState

```
Class contador extends Component{  
  constructor(props){  
    super(props);  
    this.state = {  
      valor:props.valor,  
    }  
  }  
  incrementar(){  
    this.setState({  
      valor: this.state.valor + 1  
    });  
  }  
}
```

En todos los métodos que **no sean** el **constructor** debes utilizar **this.setState()**.

REACT setState

```
Class contador extends Component{  
  //Aquí va el constructor  
  incrementar(){  
    this.setState({  
      valor: this.state.valor + 1  
    });  
  }  
  render(){  
    return (  
      <button  
        onClick={this.incrementar}>  
      </button>  
    );  
  }  
}
```

Con el evento **onClick** vamos a estar modificando a través del método **incrementar**, el estado de nuestro componente.

REACT setState

Para saber más sobre setState en los componentes statefull puedes acceder a la documentación oficial de React haciendo click en el siguiente [Link](#):



REACT EVENTOS

Los **eventos** son manejados a través de **métodos** que van a guardar la lógica que realizará una operación cuando el usuario interactúe con el componente.



REACT Métodos

Dentro de la clase, es decir, nuestro componente, definimos los métodos que contendrá la lógica a ejecutarse cuando se ejecute el evento.

Para definir un método, usamos la misma sintaxis que al momento de escribir una función, pero sin la palabra `function`.

```
{}
cambiarColor () {
  // aquí escribimos el código
}
```

REACT Eventos

Los eventos en React son los mismos que en JavaScript. Para hacer uso de los mismos hay que tener en cuenta que:

- ▶ se escriben en la etiqueta, como si fuera un atributo
- ▶ se nombran usando la palabra `on` seguida del nombre del evento, usando siempre camelCase
- ▶ se usan las llaves y la palabra reservada `this` para asociarlo con el método que queremos

```
{}
<button
  className="btn"
  onClick = { this.cambiarColor }>
  Cambiar Color
</button>
```

3

Cualquier componente puede utilizar eventos. Ahora bien, si el evento **modifica** el estado del componente, el mismo debe ser un **componente stateful**, es decir, con estado.



{ código }

```
class Contenedor extends Component {
  saludar () {
    alert('¡Bienvenida/o al sitio!');
  }
  render () {
    return (
      <div onMouseOver={this.saludar}>...</div>
    );
  }
}
```

5

2

4

6

{ código }

```
class Contenedor extends Component {  
    saludar () {  
        alert('¡Bienvenida/o al sitio!');  
    }  
    render () {  
        return (  
            <div onMouseOver={this.saludar}>...</div>  
        );  
    };  
}
```

Dentro de la clase y antes del render(), definimos el **método** saludar.

{ código }

```
class Contenedor extends Component {  
    saludar () {  
        alert('¡Bienvenida/o al sitio!');  
    }  
    render () {  
        return (  
            <div onMouseOver={this.saludar}>...</div>  
        );  
    };  
}
```

Al <div> le asigno el evento onMouseOver escribiendo el nombre como si fuera un atributo de la etiqueta.

7

8

{ código }

```
class Contenedor extends Component {  
    saludar () {  
        alert('¡Bienvenida/o al sitio!');  
    }  
    render () {  
        return (  
            <div onMouseOver={this.saludar}>...</div>  
        );  
    };  
}
```

Al evento le **vinculo** el método saludar que defini previamente. Cuando el usuario pase el mouse por el <div>, se disparará el método.

9

10

REACT Eventos

Para saber más sobre los eventos puedes acceder a la documentación oficial de React haciendo click en el siguiente [Link](#):



REACT

Función del evento

En la mayoría de los casos al evento no le enviamos parámetros.

```
class Navbar extends Component {  
    saludar () => {  
        alert()  
    }  
  
    render() {  
        return (  
            <button onClick={this.saludar}>  
                Cambiar Color  
            </button>  
        );  
    }  
}
```

11

REACT

Función del evento

En la mayoría de los casos al evento no le enviamos parámetros.

```
class Navbar extends Component {  
    cambiarColor () => {  
        // aquí escribimos el código  
    }  
  
    render() {  
        return (  
            <button onClick={this.cambiarColor}>  
                Cambiar Color  
            </button>  
        );  
    }  
}
```

12

REACT

Parámetro event

Sin embargo react siempre entrega el evento como único parámetro a la función. Allí podemos saber, entre otras cosas, cuál es el elemento que emitió el evento.

```
class Navbar extends Component {  
    cambiarColor (event) => {  
        console.log(event);  
    }  
  
    render() {  
        return (  
            <button onClick={this.cambiarColor}>  
                Cambiar Color  
            </button>  
        );  
    }  
}
```

13

REACT

Parámetros del evento

Si quisieramos pasarle un parámetro al evento, envolvemos la función dentro de otra. Para pasar ambos datos, tomamos el evento y lo pasamos a la función junto con la otra información

```
class Navbar extends Component {  
    cambiarColor = (event, dato) => {  
        console.log(event, dato)  
    }  
  
    render(){  
        return (  
            <button  
                onClick={(e)=>{this.cambiarColor(e,dato)}}>  
                Cambiar Color  
            </button>  
        );  
    }  
}
```

14

REACT

Múltiples funciones

```
class Navbar extends Component {  
    primero = () => console.log('primero')  
    segundo = () => console.log('segundo')  
    hacerTodo = () => {  
        this.primero();  
        this.segundo();  
    }  
    render(){  
        return (  
            <button onClick={this.hacerTodo}>  
                Cambiar Color  
            </button>  
        );  
    }  
}
```

15

¿Cómo?

Los eventos en React se nombran usando camelCase, en vez de minúsculas.

Los eventos se definen en la etiqueta de la misma manera que los atributos.

```
<button  
    className="btn"  
    onClick={this.cambiarColor}>  
    Cambiar Color  
</button>
```

Recordá que cuando hagas la llamada a los eventos, éstos van entre llaves y **no** llevan paréntesis.

16

REACT

Crear un componente stateful

Cualquier componente puede utilizar eventos. Si el evento modifica el estado del componente, éste debe ser stateful.

```
{ } import React, {Component} from "react";  
  
class Navbar extends Component {  
    render(){  
        return (  
            <button onClick={this.cambiarColor}>  
                Cambiar Color  
            </button>  
        );  
    }  
}  
  
{ } Export default Navbar;
```

17

Función del evento

En la mayoría de los casos al evento no le enviamos parámetros

```
class Navbar extends Component {  
    cambiarColor = () =>{  
        //Aquí escribimos el código  
    }  
  
    render(){  
        return (  
            <button onClick={this.cambiarColor}>  
                Cambiar Color  
            </button>  
        );  
    }  
}
```

18

Parametros del evento

Sin embargo react siempre entrega el evento como unico parametro a la función. Allí podemos saber, entre otras cosas, cual es el elemento que emitió el evento.

```
class Navbar extends Component {  
    cambiarColor = (event) =>{  
        console.log(event)  
    }  
  
    render(){  
        return (  
            <button onClick={this.cambiarColor}>  
                Cambiar Color  
            </button>  
        );  
    }  
}
```

19

Parametros del evento

Y si quisieramos pasarle un parametro al evento, envolvemos la función dentro de otra. Para pasar el evento y otro dato, tomamos el evento y lo pasamos a la función junto con la otra información

```
class Navbar extends Component {  
    cambiarColor = (event, dato) =>{  
        console.log(event, dato)  
    }  
  
    render(){  
        return (  
            <button  
                onClick={(e)=>(this.cambiarColor(e,'dato'))}>  
                Cambiar Color  
            </button>  
        );  
    }  
}
```

20

Multiples funciones

Y si quisieramos pasarle un parametro al evento, envolvemos la función dentro de otra. Para pasar el evento y otro dato, tomamos el evento y lo pasamos a la función junto con la otra información

```
class Navbar extends Component {  
    primero = () => console.log('primero')  
    segundo = () => console.log('segundo')  
    hacerTodo = () =>{  
        this.primero();  
        this.segundo();  
    }  
    render(){  
        return (  
            <button onClick={this.hacerTodo}>  
                Cambiar Color  
            </button>  
        );  
    }  
}
```

21

REACT CICLO DE VIDA

Todo componente con **estado** tiene varios **métodos de ciclo de vida** que se pueden sobrescribir para ejecutar código en momentos particulares del proceso.



componentDidMount()

Es un método que sólo se ejecuta en el cliente y se produce inmediatamente después del primer **renderizado** del componente.



{ código }

```
componentDidMount (){
    // código
}
```

Una vez que se ejecuta este método, quedarán disponibles los elementos asociados al componente dentro del DOM.

Si quisieramos hacer un llamado asíncrono, éste sería el método en el que deberíamos definirlo.

componentDidUpdate()

Se ejecutará cada vez que el componente sufra algún **cambio** de **estado**, provocado por sí mismo o por un componente padre.



{ código }

```
componentDidUpdate (prevProps, prevState){
    // código
}
```

Este método **puede recibir dos parámetros**, ambos objetos literales, representando las propiedades previas y el estado previo del componente.

{ código }

componentWillUnmount()

Se invoca automáticamente antes de desmontar y destruir un componente.

Dentro de este método no se debe invocar el método `setState`, ya que al pasar por el método de desmontaje, nunca más volverá a ser renderizado.



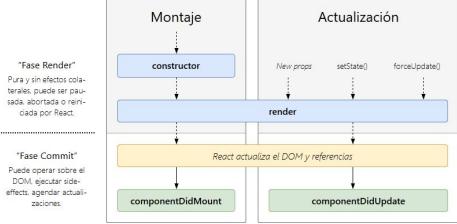
```
componentWillUnmount (prevProps, prevState){  
  // código  
}
```

Este método es llamado justo antes de que el componente sea removido del DOM.

Sirve para hacer cualquier operación de limpieza, tales como invalidar timers, eliminar elementos que se hayan creado durante `componentDidMount()`, o cualquier otra operación pendiente.

REACT

Fases del Ciclo de vida



9

REACT

Ciclo de vida

Para saber más sobre ciclo de vida de los componentes statefull puedes acceder a la documentación oficial de React haciendo click en el siguiente [Link](#):



10

REACT INTEGRACIÓN CON APIs

Con React, puedes usar cualquier biblioteca AJAX. Algunas de las más populares son **Axios**, **jQuery AJAX**, y **window.fetch**, la cual es soportada de manera nativa en la mayoría de navegadores modernos.



REACT Montado del componente

Se invoca esta función una vez ya ejecutado el método render().

Como el DOM ya es accesible podemos en este método realizar cualquier manipulación sobre él.

Se estila en este método hacer pedidos a endpoints via AJAX, inicializar timers o generar suscripciones a servicios.

```
componentDidMount(){
  fetch(URL)
    .then((r)=>console.log(r))
    .catch((e)=>console.log(e));
}
```

Deberías ejecutar tus llamadas AJAX en el ciclo de vida **componentDidMount**. De esta manera, podrás llamar a **setState** para actualizar el componente una vez que hayas recibido tus datos.

3

REACT Integración con APIs

Para saber más sobre integración con APIs puedes acceder a la documentación oficial de React haciendo click en el siguiente [Link](#):

