CLASE 1: Introducción a HTML

Página, Sitio y Aplicación

Página web | es un documento HTML, pueden ser estáticas o dinámicas.
Sitio web | un conjunto de páginas web estructuradas en un dominio.
Aplicacion web | software desarrollado con tecnología web. Está instalada en un Servidor y se accede mediante una URL (dirección) en el navegador. Ejemplo: excel, meet.

Front-End

Es la parte visible del usuario (cliente) de una página web.
HTML, CSS y JavaScript.

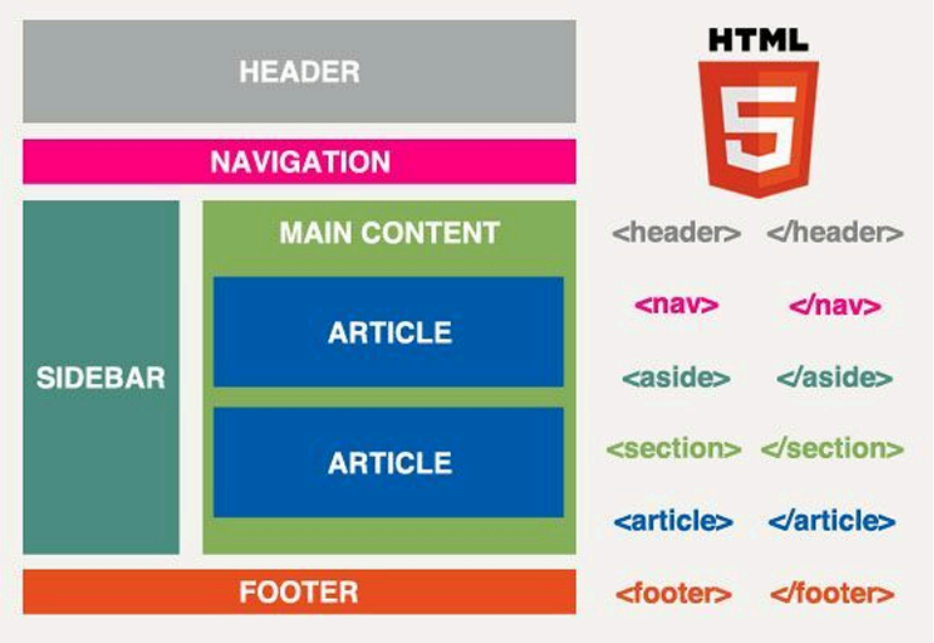
Back-End

Corre en el servidor.
Manejo de algoritmos y utilización de base de datos para guardar o procesar información.

HTML

Define la estructura, semántica y contenido de las páginas web.
Utiliza etiquetas (tags) que definen la estructura del documento.
Texto plano con extensión **.html**

Estructura genérica de una página web



Para generar un comentario: `<!-- Esto es un comentario →`

Etiquetas semánticas

- **section** | define una sección de contenido monotemático.
- **article** | define un fragmento de información dentro de una sección.
- **header** | define la cabecera de un contenido determinado o del documento.
- **footer** | define el pie de un contenido determinado o del documento.
- **nav** | define un apartado de navegación.
- **main** | define contenido principal del body.
- **aside** | agrupa contenido no relacionado con el tema principal del documento.

CLASE 2: Etiquetas, listas, hipervínculo e imagen

Etiquetas de texto

- Elementos de encabezado **<h1>** | Tiene seis niveles h1…h6. h1 solo debe ser utilizado una vez por documento HTML.
- Elementos de Párrafo **<p>** |

Listas

- **Ordenadas (ol)** | **> li**. Se puede cambiar el tipo de viñeta usando el atributo **"type"** y cambiando su valor. **1** para números, **A** para alfabeto e **I** para números romanos. También se puede usar el atributo **start** para definir dónde empieza la numeración.
- **Desordenadas (ul)** | **> li**. Tipos de viñetas:
  - bullet, circle, square, upper-roman, lower-alpha
- **De definicion (dl)** | dentro tienen <dt> (terminó) y <dd> (definición del término)
- **Listas Anidadas** | permite crear varios niveles de jerarquía y organización.

Rutas (path) | Es una dirección o camino, que permite encontrar un recurso.

- **Ruta Absoluta** | por ejemplo un sitio web.
- **Ruta Relativa** | acceder a un recurso desde mi posición actual. Ejemplo: ../imgs/img1.jpg

Enlaces

A través de la etiqueta **<a>** se puede crear los enlaces.

- **Externos** | es una **ruta absoluta**. youtube
  - **target**="_blank" | para abrirlo en una pestaña nueva.
  - **rel**="noopener noreferrer nofollow"
- **Locales** | **ruta relativa** siempre que sea posible. inicio
- **Anclas** | referencia a una determinada parte de la página, inician con # en el href (id)
- **Correo** | al hacer clic en ellos, se abrirá el programa de correo predeterminado.
- **Teléfono** | listo para llamar.
- **Combinados** | por ejemplo href="./pagina2.html#tema2" (ancla de otro archivo html)
- **Archivos** | por ejemplo **<a** href="**archivo.pdf" download**>Descargar PDF****

⚠️ **para subir un nivel de carpeta se usa "../"**

Multimedia con HTML

Imágenes <img>

Esta etiqueta requiere de dos atributos obligatorios:

- **src** | fuente (ruta)
- **alt** | para posicionamiento en buscadores, personas con dificultades visuales y para cuando la imagen no se encuentra disponible.

Con **width** y **height** podemos definir el ancho y alto de la imagen.

Podemos utilizar una imagen como enlace combinando las etiquetas **<a>** e **<img>**.

Favicon

Es una pequeña imagen que se muestra en la pestaña del navegador o en la lista de marcadores.
Tamaño de 16x16 píxeles. Debe estar en formato **.ico** y va en el head del HTML.
link:favico que va en el **head**. https://convertio.co/es/

CLASE 3: Multimedia, Introducción a CSS

<iframe> | insertar contenido de otros sitios webs dentro de nuestro sitio.
<video> | controls, poster, autoplay, loop, muted, preload, src, type, width y height.
<audio> | preload, src, controls, type, autoplay, loop, muted.

CSS (Hoja de Estilo en Cascada)

Está compuesta de **reglas**, **selectores** y **declaraciones**.
Sirven para estilizar nuestro contenido HTML.

Métodos para vincular CSS

- ❌ **Interna** | a través de la etiqueta <style> dentro del <head>
- ❌ **En línea** | usando el atributo style en cada elemento del HTML.
- ✅ **Externa** | escribimos todos los estilos en un archivo **.css** y vinculandolo con la etiqueta **<link>** dentro del **<head>. < link href="css/estilos.css" rel="stylesheet" >**

Selectores de CSS

UNIVERSAL * | selecciona todos los elementos de html.
ID | se usa el #nombreId. Se recomienda usar nombres únicos.
CLASE | se usa el .nombreClase
ETIQUETA | se usa el nombre de una etiqueta específica.
COMBINADOS | por ejemplo p.nombreClase
DESCENDIENTES | sirven para agregar especificidad, se utilizan con un espacio.

Sintaxis CSS

Formatos de color

- **Hexadecimal** | #f1f1f1
- **RGB** | rgb(240, 83, 21)
- **RGBA** | gba(240, 83, 21, 0.3) [el último representa la opacidad]

Propiedades para establecer el fondo de cualquier elemento:

- **background-color** | establecer un color de fondo del elemento.
- **background-image** | permite asignar una imagen de fondo al elemento.
- **background-repeat** | para controlar si se va a repetir y dispuesta.
- **background-attachment** | establece si la img se va a mover junto con la página.
- **background-position** | el primero para especificar la posición en el eje x y el segundo en el eje y.
- **background-size** | establecer el tamaño de la imagen de fondo.

Especificidad: !important > inline > ID > Clases > Etiquetas

CLASE 4: Modelo de caja

Pseudo selectores

Nos permite controlar eventos especiales de un elemento.
Suelen ser aplicados sobre un selector existente.

Sintaxis | selector:**pseudoselector** { propiedad: valor; }

:hover | Solo será visible al posar el cursor sobre ese elemento.
:focus |Solo será visible al momento de clickear dentro del campo.

Elementos de línea y de bloque

Etiquetas de bloque: intenta ocupar el 100% del ancho del sitio. **Ejemplo: **
Etiquetas en línea: ocupan sólo el ancho de su contenido. **Ejemplo **

Tipos de elementos

- inline, block, inline-block y none

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.

Modelo de caja

Es el comportamiento que hace que todos los elementos de un documento HTML se presenten mediante cajas rectangulares.
Las propiedades de modelo de caja solo se aplican a **etiquetas de bloque**.

Propiedades:

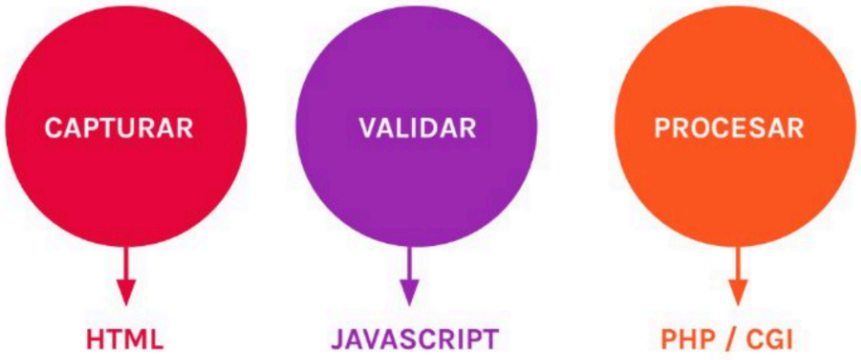
- **width** | Sí un elemento no tiene declarado la propiedad width, el ancho será igual al 100% de su padre contenedor, siempre y cuando ése sea un elemento de bloque. Asignación de valor mediante porcentajes **(%)** ó píxeles **(px)**.
- **height** | Sí un elemento no tiene declarado la propiedad height, el alto será igual a la altura que le proporcione su contenido interno, sea un elemento de bloque o de línea. (px).
- **padding** | Hace referencia al margen interior del elemento. Valor en **px**, 1 valor para los 4 lados de la caja o 2 valores (arriba y abajo - izquierda y derecha).
- **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. **> border: solid 3px green**;
- **margin** | Hace referencia al margen del elemento. Sirve para separar una caja de la otra. Valor asignado en **px** de 1 valor o 2 valores.
- **box-sizing** | Descuenta automáticamente del ancho y alto lo que agregamos de relleno y borde. El margin se sigue sumando.

Tablas

Se usan para representar datos. Se usa la etiqueta **<table>** y dentro **<tr>** que representa una fila, **<td>** representa cada celda y **<th>** representa una celda de encabezado.
Atributos: colspan y rowspan.

CLASE 5: Formularios

Un formulario es un sistema para capturar datos. Para que funcione correctamente, hacen falta las siguientes tres instancias.



Semántica

El uso correcto de nuestras etiquetas, desde el punto de vista semántico, nos permite reforzar el significado del contenido de nuestro sitio web. Ser más específico al momento de encerrar contenido entre etiquetas y así crear un código más amigable para los motores de búsqueda.

Elementos del formulario

<form>

El tag más importante, sino el formulario no funciona. Atributos: action (a donde se debe enviar este formulario), method.

<input>

Nos permite generar campos para que el usuario complete con información. Atributos: type, name.

- type="radio" | generamos un botón de única opción, conocida como "radio-button".
- type="checkbox" | generamos una casilla de verificación para que el usuario seleccione una o más opciones, más conocidos como "check boxes".
- name="variable", este input se va a guardar en una variable para el servidor.

<label>

Acompaña a un campo. Para indicar la información a completar. El atributo **for** tiene que coincidir con el atributo **id** del input, de esta forma estamos linkeando el label con el input.

<textarea>

Creamos un campo para escribir varias líneas de texto. Para enviar comentarios.

<select>

Con la etiqueta <select> y la etiqueta <option> creamos un combobox o dropdown de única selección. El <select> será el contenedor para <option>.

<button>

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.

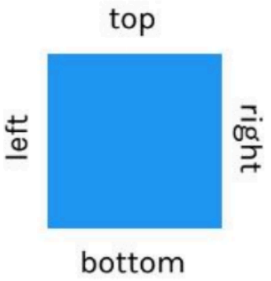
Subida a un servidor (hosting)

https://app.netlify.com/drop

Posicionamiento en CSS

El posicionamiento nos permite **trasladar un elemento** desde su posición original a una nueva posición, también nos permite superponer elementos.

↑ Arriba (top)
→ Derecha (right)
↓ Abajo (bottom)
← Izquierda (left)

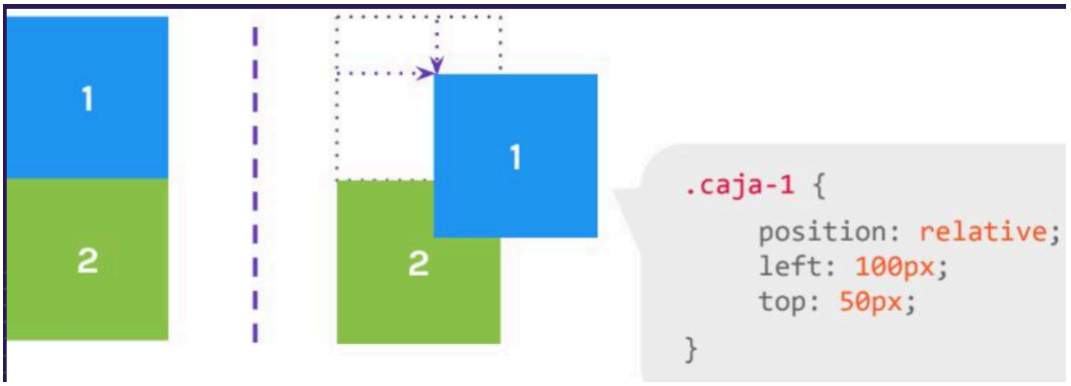


Cada uno de los elementos de nuestra página web tiene cuatro **puntos de referencia**.

Cuando desplazamos un elemento tomando un costado como referencia, el movimiento será **positivo si empujamos** el elemento y **negativo si tiramos** de él.

Posicionamiento Relativo

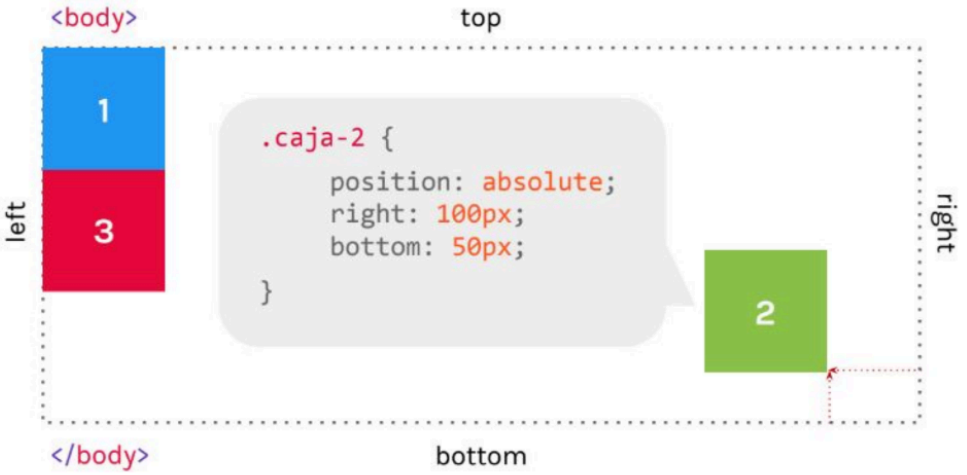
Para trasladar un elemento tomamos como referencia sus propios costados (propio elemento).



Cuando movemos una caja de manera relativa, el espacio que ocupaba originalmente la caja seguirá ocupado. Lo usamos entonces, cuando queremos desplazar un elemento sin modificar el flujo original.

Posicionamiento Absoluto

Se toma como referencia el **body**, los puntos de referencia serán los costados del body. Cuando movemos el elemento su espacio original quedará vacío. Uso común: para crear menús desplegables, ventanas modales o elementos que deben estar en una ubicación específica.

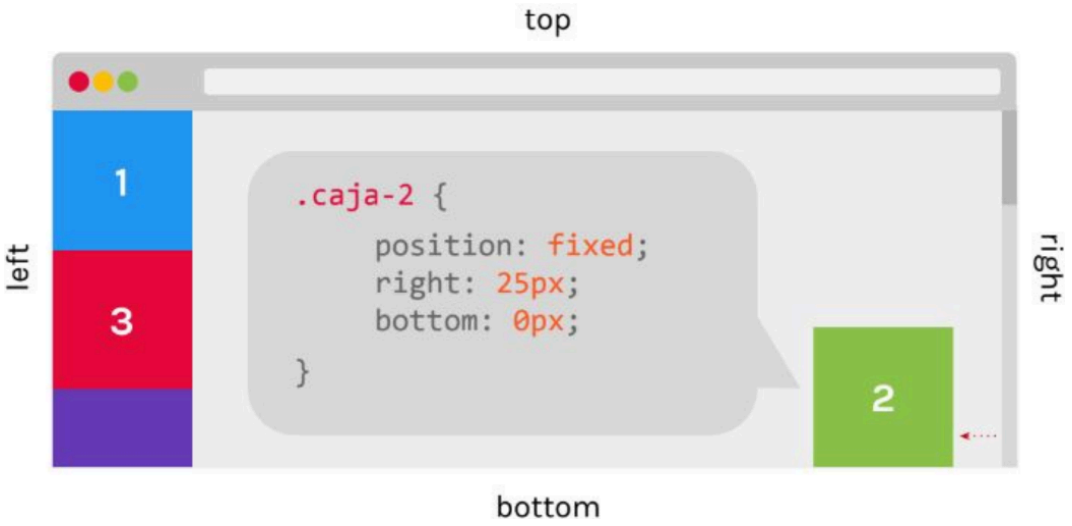


Posicionamiento Absoluto + Relativo

Si el contenedor tiene **position: relative** y un elemento hijo tiene **position: absolute**, el hijo se posiciona en relación al contenedor.

Posicionamiento Fijo

Se toma como referencia la ventana del navegador. Los puntos de referencia serán los costados de la ventana del navegador.
Sin importar que hagamos scroll en la página el elemento **siempre se mantendrá fijo** con respecto a la ventana del navegador.



Viewport

La etiqueta **< meta name= “viewport” >** es fundamental para controlar cómo se muestra la página web en dispositivos móviles. Diseñada para mejorar la experiencia en pantallas pequeñas.
La etiqueta se coloca dentro de la sección **<head>** del HTML.

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

Unidades de medida

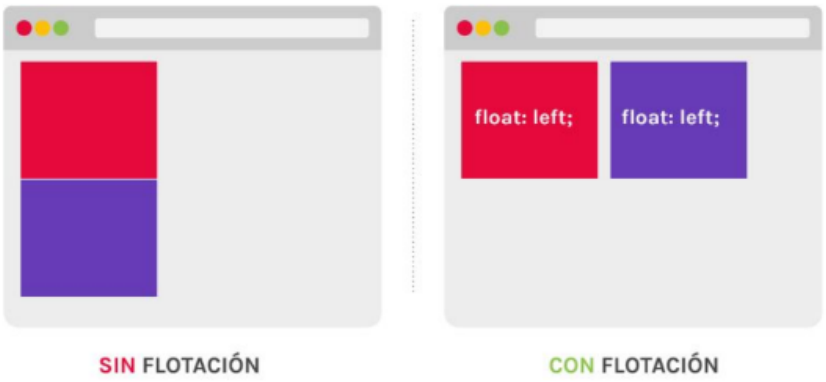
Em’s (**em**) se recomienda usar en todo lo relacionado con tipografías. 1em = 16px.
Viewport measures, Cualquier medida expresada en viewport width (**vw**) ó viewport height (**vh**) tomará SIEMPRE como eje de referencia al viewport del documento.

CLASE 6: Propiedad Float

Mediante la propiedad **float**, le estamos otorgando a un elemento la habilidad de flotar. Es decir, “despegarse” de ese flujo natural del sitio, y decidir hacia dónde queremos que ese elemento flote (izquierda o derecha) dentro de su contenedor..

La propiedad float recibe cuatro valores posibles:

- **left** | El elemento flota hacia la izquierda del contenedor.
- **right** | El elemento flota hacia la derecha del contenedor
- **none** | Valor predeterminado. No se aplica flotación. El elemento permanece en el flujo normal del documento.
- **inherit** | El elemento hereda el valor de float de su elemento contenedor.



Es importante determinar un ancho para nuestras cajas. De esta forma, vamos a poder controlar cuántas de ellas entran en una misma línea. Si no se define un ancho, el elemento se expandirá hasta ocupar el ancho total disponible.
Si la suma de los anchos de las cajas supera el ancho del contenedor padre, éstas no van a entrar en la misma línea, sin importar que tengan asignada la propiedad de float.

```
.box {
  float: left;
  width: 30%; /* Asignando ancho específico */
  margin: 1%; /* Espacio entre cajas */
}
```

📌 Actualmente es más recomendable usar **flexbox** o **grid** para lograr un diseño de páginas web más robusto y adaptable. La propiedad float sigue siendo valiosa en situaciones específicas, como para alineaciones simples de imágenes o texto, pero en la mayoría de los casos de diseño de layouts completos, **flexbox** o **grid** son soluciones más eficientes y modernas.

CLASE 7: Media Queries en CSS

Son fundamentales para lograr un diseño adaptable y responsivo, ya que permiten aplicar estilos específicos según el tamaño, la resolución o la orientación de la pantalla.

Min-width (Mobile first)

Al especificar **min-width**, estamos diciendo “si como mínimo hay Npx de ancho, apliquemos estas reglas”. Similar a decir ➡️**Desde este ancho, hacia arriba.**

Max-width (Mobile last)

Al especificar **max-width**, estamos diciendo “sí como máximo hay Npx de ancho, apliquemos estas reglas”. Similar a decir ➡️**Desde este ancho, hacia abajo.**

Orientation

Permite definir estilos específicos según la orientación de la pantalla o del dispositivo.

- Portrait (vertical)
- Landscape (horizontal)

✅ Buenas Prácticas

- Escribir las media queries al final del CSS.
- Empezar con el diseño base para dispositivos pequeños y luego usar **min-width** para añadir estilos progresivos (Mobile First), o **max-width** para estilos regresivos (Mobile Last) en caso de empezar con el diseño para pantallas grandes.

Flexbox

Es una metodología de CSS que permite maquetar un sitio web utilizando una estructura de filas y columnas.
Esta metodología propone una estructura basada en el uso de un contenedor-padre (**Flex-container**) y sus elementos hijos (**Flex-items**).

Flex-container (Contenedor Flex)

Es el elemento contenedor al que se le aplica la propiedad **display: flex**. Controla la disposición de todos sus elementos hijos.
Propiedades:

- display: flex
- flex-direction | define el eje principal de disposición de los elementos.
- justify-content | controla la alineación de los elementos a lo largo del eje principal.
- align-items | controla la alineación de los elementos a lo largo del eje transversal.
- align-content
- flex-wrap

Flex-items (Elementos Flex)

Estos elementos se alinean y distribuyen automáticamente dentro del contenedor.
Propiedades:

- flex-grow
- flex-shrink
- flex-basis
- align-self

CLASE 8: Bootstrap y JavaScript

Bootstrap es una librería **open source**, trae consigo una combinación de reglas que facilitan el desarrollo de una estructura web.

✅ Vinculación externa
Para vincularlo de forma externa a nuestro proyecto tenemos que ir al sitio web oficial, copiar el link que está debajo del título CSS y pegarlo en el head de nuestro HTML.
https://getbootstrap.com/

En el head:
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhjY6hW+ALEwIH" crossorigin="anonymous">

Al final del body:
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js" integrity="sha384-YvpcrYf0tY3lHB60NNkmXc5s9fDVZLESaAA55NDzOxhy9GkcIdslK1eN7N6jIeHz" crossorigin="anonymous"></script>

❌ Vinculación Interna
Tenemos que descargar los archivos de la web oficial, de los cuales usaremos dos: **bootstrap.min.css** y **bootstrap.min.js** una vez que se tiene estos archivos hay que incluirlos dentro de las carpetas css y js de nuestro proyecto.

Contenedores y Breakpoints
Representan el núcleo de bootstrap.

Un **contenedor** es un elemento HTML, que anida otros elementos y los contiene para que se acomoden dentro del ancho y alto del mismo.

Bootstrap cuenta con dos clases definidas para generar un elemento contenedor: **container** y **container-fluid**.

container
- Ancho delimitado: se ajusta a un tamaño determinado en función del viewport, como máximo llegará a 1140 px.
- Padding de 15 px de ambos costados.
- Propiedades margin-left y margin-right con valor auto para que el contenedor esté siempre centrado en la pantalla.

container-fluid
- Lo único que cambia respecto al **container** es que el ancho es ilimitado (siempre ocupará el 100% del viewport)

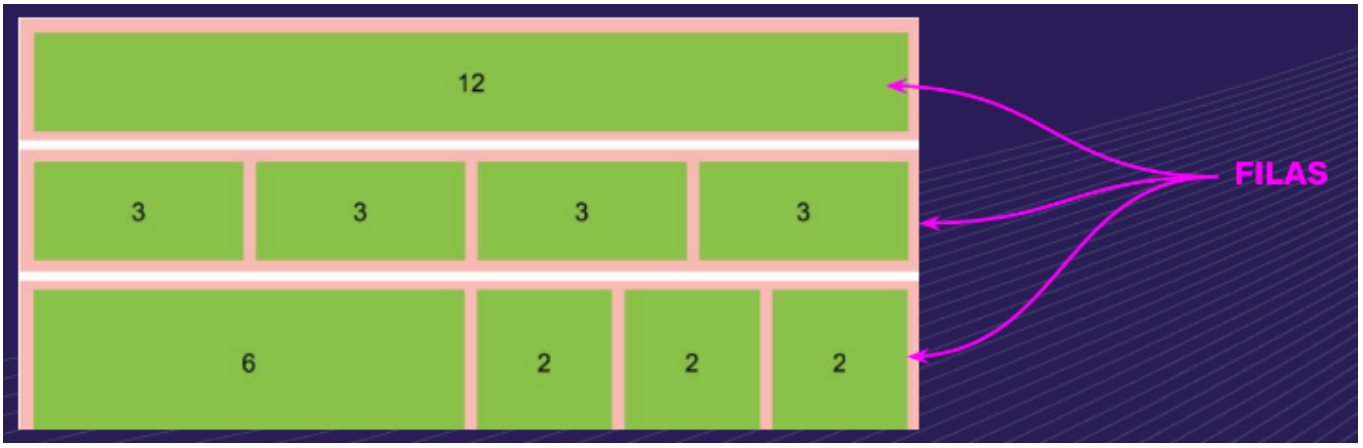
Un **breakpoint** es un punto de quiebre a partir del cual nuestro sitio cambia de diseño según el tamaño de la pantalla. Bootstrap ya tiene pre-definidos su conjunto de breakpoints.

Modelo de grilla

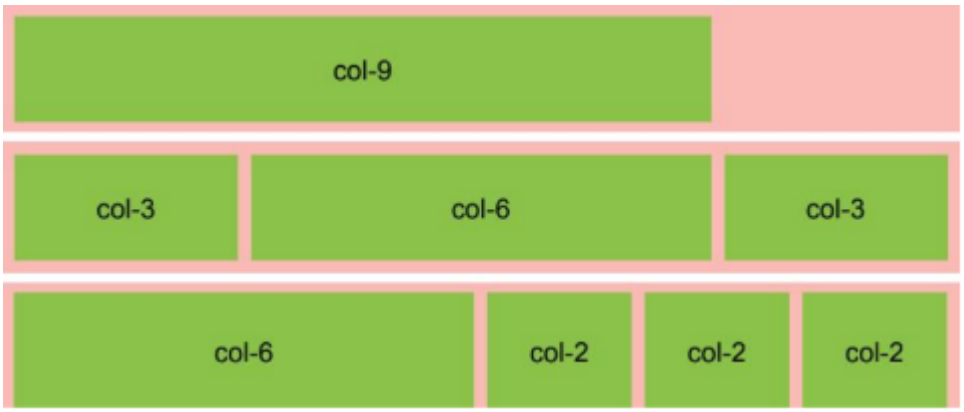
El sistema de grillas de Bootstrap está compuesto por **12 columnas**, sobre la cual diseñaremos y definiremos el flujo de todo nuestro sitio. Este sistema está construido con flexbox y es totalmente responsive.

Las **filas** son los padres contenedores de las columnas.
El ancho mínimo de una **columna** es de 1 y el máximo es de 12.

Cada fila permite un nuevo conjunto de columnas, nos da más flexibilidad para nuestro diseño.



Bootstrap nos ofrece las siguientes clases para diagramar las columnas.
row ➡ crea una fila
col ➡ crea una columna (una vez creada la fila). Se le puede asignar un tamaño específico, utilizando la clase **col-x**, donde **x** será el número de columnas que queremos ocupar.



Paso a paso:

- Crear un **contenedor**
- Crear una **fila**
- Crear las **columnas**
- Opcionalmente asignarle a cada columna su tamaño
- Opcionalmente asignarle a cada columna los breakpoints

Agregando nuestro estilo

Teniendo en cuenta las reglas de CSS debemos colocar nuestro archivo CSS después del enlace de bootstrap para sobrescribir algunas configuraciones predeterminadas de Bootstrap.

JavaScript

En Javascript existen tres tipos de variables: **var**, **let** y **const**.
El nombre de una variable solo puede estar formado por letras y los símbolos $ y _ (guión bajo).
✅Buena práctica: los nombres de las variables usen el formato camelCase.

La diferencia entre **var** y **let** es que **let** sólo será accesible en el bloque de código que fue declarado.
Const funciona igual que la variable let.

Tipos de datos	Tipos de datos especiales
number	NaN (not a number)
string	null
boolean	undefined
object	
array	

Operadores

- **de asignación** | =
- **aritméticos** | + - * / ++ %
- **de comparación simple** | ==!= > >= < <=
- **de comparación estricta** | === !==
- **lógicos** | and, or y not (&& || !)
- **de concatenación** | +

CLASE 9: Condicionales

IF - ELSE IF - ELSE

IF TERNARIO | condición **?** expresiónTrue **:** expresiónFalse ;

SWITCH | **switch** (expresión) {
```
        case valorA:
            // código que se va a ejecutar si valorA es verdadero
            break;
        default:
            // código que se ejecuta si ningun caso es verdadero
        }
```

WHILE y DO-WHILE

FOR | **for** (inicio ; condición ; modificador)

FUNCIONES DECLARADAS | **function nombre** (**parámetros**) {
Cuerpo }

FUNCIONES EXPRESADAS | **let variable = function** (**parámetros**) {
Cuerpo }

CLASE 11: Arrays, Objeto, POO

Arrays

Los **arrays** nos permiten generar una colección de datos ordenados. Se puede almacenar elementos de distintos tipos de datos.

Métodos de arrays

- **push()** | Agrega uno o varios elementos al **final del array**. Recibe uno o más elementos como parámetros. Retorna la nueva longitud del array.
- **.unshift()** | Agrega uno o varios elementos al **principio del array**. Recibe uno o más elementos como parámetros. Retorna la nueva longitud del array.
- **.pop()** | Elimina el **último** elemento de un array. No recibe parámetros y devuelve el elemento eliminado.
- **.shift()** | Elimina el **primer** elemento de un array. No recibe parámetros y devuelve el elemento eliminado.
- **.join()** | Une los elementos de un array utilizando el separador que le especifiquemos. Recibe un separador (string), opcional. Retorna un string con los elementos unidos.
- **.indexOf()** | Busca en el array el elemento que recibe como parámetro. Recibe un elemento a buscar. Retorna el primer índice donde encontró lo que buscábamos. Si no lo encuentra, retorna un -1.
- **.lastIndexOf()** | Similar a **.indexOf** solo que empieza por el final del array.
- **.includes()** | También similar a **.indexOf**, con la salvedad que retorna un booleano.

CALLBACK

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

- **.map()** | Este método recibe una función como parámetro (callback). Recorre el array y devuelve un nuevo array modificado.
- **.filter()** | Este método también recibe una función, recorre el array y filtra 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.
- **.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.
- **.forEach()** | La finalidad de este método es iterar sobre un array. Recibe un callback como parámetro y no retorna nada.

```
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]
```

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

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

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

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

Objetos Literales

Un **objeto literal** es una estructura que permite almacenar datos en pares de **clave-valor**. Es muy útil para representar entidades con **propiedades** y **métodos**. Los objetos pueden contener cualquier tipo de valor, incluidos otros objetos, arrays y funciones.
Con la notación **objeto.propiedad** accedemos al valor de dicha propiedad. Si una propiedad almacena una función lo llamamos método del 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.
Con la notación **this.propiedad** accedemos al valor de cada propiedad interna de ese objeto.

CONSTRUIR UN OBJETO CON UNA FUNCIÓN CONSTRUCTORA
La **función constructora** funciona como plantillas para crear objetos similares. Permite instanciar múltiples objetos con las mismas propiedades y métodos. El nombre de la función será el nombre de nuestro objeto, por convención con la primera letra en mayúscula.
Con la palabra reservada **new** podemos instanciar un objeto.

Spread operator (...)

El operador de propagación **(...)** permite expandir un iterable (como un array o un objeto) en elementos individuales. Tiene varios usos importantes:

- **Spread en arrays** | para copiar todos los datos de un array y crear uno nuevo. También se puede agregar todos los datos del array dentro de un array existente.
- Implementando este operador, podemos pasarle a una función un array como argumento. Por ejemplo **Math.min**(n parámetros) implementando este spread > Math.min(...nums).

-
- **Spread en objetos** | Permite copiar propiedades de un objeto en un objeto nuevo. Por ejemplo, tengo el objeto persona con las propiedades nombre y edad, creo otro objeto con esas mismas propiedades (...persona) y le puedo adicionar propiedades propias.
-
- **Rest Parameter** | Utilizado como último parámetro de una función nos permite capturar cada uno de los **argumentos adicionales** pasados a esa función. En este caso se utiliza durante la definición de la función y no durante la ejecución. Esto es útil para manejar un número versatil de argumentos.

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

Diferencia entre Spread y Rest:

- **Spread (...)** expande un iterable en elementos individuales, se usa en arrays y objetos.
- **Rest (...)** agrupa múltiples argumentos en un solo array y se usa en la definición de funciones.

Destructuring

Permite extraer datos de arrays y objetos literales de una manera más sencilla y fácil de implementar. No modifica el array u objeto literal de origen.

Desestructurando Arrays

Declaramos una variable y entre corchetes, escribimos el nombre que queremos y si es más de una variable los separamos con una coma. Partimos de un array previamente definido y se transfiere cada dato a las variables que definimos.

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

Si queremos saltar un valor, podemos dejar vacío el nombre de la variable que corresponde a esa posición. Ejemplo: let [**color1** , , **color2**] = colores ;

Desestructurando Objetos

Para desestructurar un objeto literal, creamos una variable, y entre llaves, declaramos el o los nombres de las propiedades que queremos extraer. Partiendo de un objeto previamente definido, si en algún caso necesitamos cambiarle el nombre a la variable que estamos creando se debe colocar dos puntos a continuación de la propiedad

```
let persona = {nombre: 'Cristian' , edad: 30 , faltas: 3 } ;
let {nombre, edad, faltas: totalFaltas } = persona;
```

POO

Es un paradigma de programación que organiza el software en torno a “objetos”. Consiste en abstraer los elementos importantes que conforman objetos (cosas) del mundo real en código.

Conceptos claves:

- **Clase**: plantilla para crear objetos, las clases se encargan de encapsular las propiedades y funciones.
- **Objeto**: es una instancia de una clase.

Pilares de la POO

- **Abstracción** | Eliminar los detalles innecesarios.
- **Encapsulamiento** | Ocultar los detalles (del funcionamiento) irrelevantes para el exterior.
- **Herencia** | forma de heredar métodos y atributos de otra clase (superclase o padre).
- **Polimorfismo** | realizar una misma acción con diferente resultado.

Modificadores de acceso: public y private (en este caso se usa el hashtag adelante #propiedad).

Para acceder al valor de una propiedad privada usamos un **getter**.
Para modificar una propiedad privada usamos un **setter**.

CLASE 12: DOM

Vinculación interna y externa del HTML con JS.

- **Buena práctica**: poner la vinculación con el archivo js **al final** del body de nuestro HTML.

DOM (Document Object Model)

- **Objeto window**: representa la ventana donde estamos navegando.
- **Objeto Document**: representa al HTML. El documento se carga dentro del objeto window.

Interacción con el usuario: alert(), confirm(), prompt().

Selectores: cada selector puede retornar un solo elemento o una lista de elementos. Son métodos del objeto document.

- **querySelector()**; retorna el primer elemento que coincida
- **querySelectorAll()**; retorna un listado de elementos que coincidan con la búsqueda especificada.
- **getElementById()**; recibe un string con el nombre del id del elemento del DOM.

Modificando el DOM: se tiene que tener seleccionado el objeto a modificar.

- **innerHTML:** para leer o modificar **contenido HTML**, incluyendo etiquetas y estructuras.
-
```
document.querySelector("div.nombre").innerHTML;
```
```
document.querySelector("div.nombre").innerHTML = "Leon";
```
- **innerText:** para leer o modificar el texto (plano) que posea un elemento HTML osea solo el texto visible ignorando las etiquetas HTML.

Modificando estilos: usando la propiedad style, que nos permite leer y sobreescribir las reglas CSS.

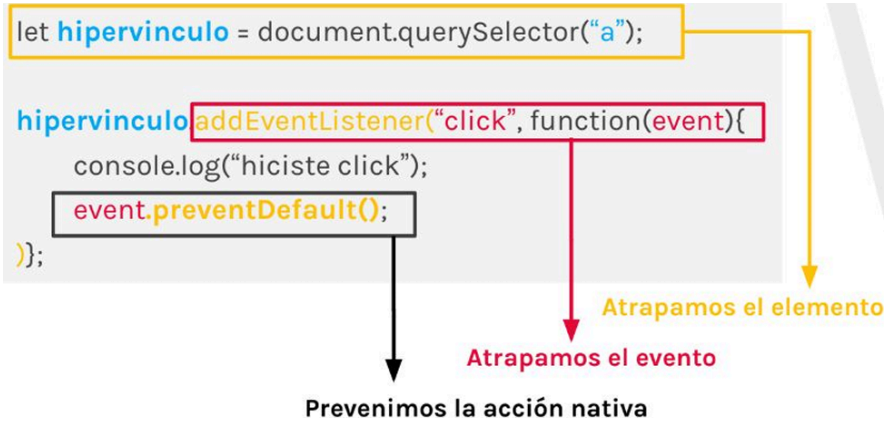
Modificando clases: con el atributo class de un elemento.

- **classList.add()** | permite agregar una clase nueva al elemento seleccionado.
- **classList.remove()** | permite quitar una clase existente al elemento.
- **classList.toggle()** | agrega o remueve una clase, revisando si existe.
- classList.contains() | permite verificar una clase especifica, devuelve un booleano.

Eventos

- eventos más usados: onclick, onscroll, onload, etc.
- **onload**: permite que todo el script se ejecute cuando se haya cargado por completo el *objeto document* dentro del *objeto window*.
- **preventDefault()** | nos permite evitar que se ejecute el evento predeterminado o nativo del elemento.
- aplicamos el evento con un **.on"evento"** o con un **addEventListener("evento", function() {}) ;**
- **eventos de teclado**:
    - **onkeydown** | Se lanza cuando una tecla se presiona, para teclas que producen o no un carácter.
    - **onkeyup** | El evento se inicia cuando la tecla es soltada.
    - **onkeypress** | Se dispara al finalizar el recorrido de la tecla.

preventDefault()



on"evento" o addEventListener()

MOUSEOVER

```
let texto = document.querySelector(".text");

texto.onmouseover = function(){
    console.log("pasaste el mouse");
}
```

También podríamos hacer...

```
texto.addEventListener("mouseover", function(){
    console.log("pasaste el mouse");
});
```

Timers

- **setTimeout** | cuando queremos que el código se ejecute una sola vez.
- **setInterval** | cuando queremos que se ejecute una y otra vez.
- **clearTimeout** y clearInterval | para detener la ejecución del timer.

CLASE 13: Validaciones

Validaciones en formulario

- **validando un campo vacío**: usamos el método **trim()** para eliminar espacios en blanco al inicio y final del valor.
- **Array de errores**: almacenarlos para gestionarlos de manera más eficiente.
- Mostrar errores en nuestro HTML.

CLASE 14: API

Def.: Es un conjunto de reglas, definiciones y protocolos que permiten que diferentes aplicaciones, servidores o sistemas se comuniquen entre sí.

APIs Públicas
No necesitan registrarse, ni solicitudes para usarlas. Ejemplo: RESTCountries.
APIs Semi-Públicas
GYPHY o Marvel necesitamos registrarnos.
APIs Privadas
Los endpoints no están disponibles para nuestro consumo.

Endpoint
Hace referencia al punto de conexión apuntado para obtener información, osea URLs.

Object location
Se escribe location en nuestro archivo js y viene con métodos y atributos muy interesantes.

- **location.href** | devuelve toda la url.
- **location.reload()** | recarga la página.
- **query string** | Es una parte de la URL que contiene información adicional enviada al servidor, generalmente utilizada para pasar parámetros. compuesta por pares “clave=valor” separados por una &. Ejemplo: https://www.ejemplo.com/productos**?categoria=libros&precio=20-50**
- **location.search** | devuelve el query string completo (incluyendo el signo ?)
- **URLSearchParams** | una interfaz que facilita trabajar con los parámetros del query string.

```
// Ejemplo de obtener el query string completo
console.log(location.search); // Devuelve "?categoria=libros&precio=20-50"

// Trabajando con parámetros específicos (GET y HAS)
const params = new URLSearchParams(location.search);
console.log(params.get("categoria")); // "libros"
console.log(params.get("precio"));    // "20-50"
console.log(params.has(“categoria”)); // true
```

Componentes claves:

- **Request** (solicitud): del cliente al servidor utilizando los **métodos de petición HTTP**.
  - **GET:** para pedir información específica al servidor.
  - **POST:** se utiliza para enviar datos al servidor (más seguro que get).
  - **PUT/PATH:** reemplaza totalmente/parcialmente la información del recurso en el servidor.
  - **DELETE:** borra un recurso presente en el servidor.

- **Response** (respuesta): incluye un **código de estado HTTP** que indica el resultado de la solicitud y el **cuerpo** (en formato JSON o XML)

Tipos de APIs:

- **APIs Web**: permite la interacción entre servidores y aplicaciones.
- **APIs de bibliotecas**: como JQuery o React para manipular el DOM.
- **APIs de hardware**: comunicación entre las aplicaciones y el hardware del dispositivo.

Asincronía en las solicitudes API: el código no se detiene o bloquea mientras espera la respuesta a una solicitud API.

- función asíncrona.

Promesa

Las promesas son funciones que permiten manejar las operaciones asíncronas (como una llamada a una API). **Estados de una promesa**:

- **pendiente** (pending): la promesa comienza en estado “pendiente”.
- **resuelta** (fulfilled): la operación se completó con éxito y se obtiene un valor que se puede manejar utilizando **.then()**.
- **rechazada** (rejected): la operación falla por un error o no se obtiene el valor deseado, para captarlo y manejarlo utilizamos el método .**catch()**.

Formas de trabajar con una API:

- **fetch()** | recibe como 1er parámetro la URL del endpoint al cual estamos haciendo el llamado asincrónico. Devuelve una promesa. Desventajas: no maneja automáticamente errores HTTP (como 404 o 500)
    - **1er then**: encargado de recibir un callback y retorna la respuesta en formato json.
    - **2do then**: recibe un callback, el cual hará lo que pidamos con la respuesta obtenida en formato json.
- **async/await** | sintaxis que simplifica el uso de promesas.
- **Axios** | biblioteca de JS para realizar solicitudes HTTP, utiliza promesas con manejo automático de errores de estado HTTP y del JSON. Se linkea el CDN. Tambien se puede instalar via npm “npm install axios”
- otras formas: **XMLHttpRequest**, **jQuery AJAX**, **WebSocket**.

Async y **await** son palabras claves en JS que permiten escribir código asincrónico de forma más legible y clara.
Cuando definimos una función con **async** delante, automáticamente esta función devuelve una promesa. **Await** se utiliza dentro de una función async y espera el resultado de una promesa sin bloquear el resto del código. Si la promesa es rechazada, arroja un error que puede ser capturado con **try…catch**.

Ventajas:

- **Legibilidad**: el código asincrónico se ve y actúa como código sincrónico.
- **Control de errores**: de manera más clara con try-catch que con **.catch()**
- **Evita el “callback hell”**: reemplaza las largas cadenas de **.then()**

CLASE 15: Arquitectura Cliente-Servidor, Node.Js

CLIENTE: Son los dispositivos que hacen peticiones de servicios o recursos a un servidor a través de un navegador web.
SERVIDOR: Es el equipo que brinda los servicios y recursos a los que acceden los clientes.

FLUJO CLIENTE/SERVIDOR
Request: es la solicitud que hacemos a través del navegador (el cliente) a un servidor.
Response: el servidor recibe nuestra solicitud, la procesa y envía una respuesta al cliente.

Esto lo debemos relacionar con los frentes del desarrollo web: **front-end** y **back-end**.

FRONT-END | es todo lo que pasa del lado del cliente (en el navegador). 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. Se incluye todo el funcionamiento interno y lógico 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 y también Node.js en entornos modernos.

Node.js

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

Componentes clave de Node.js:

- **NPM**
- **Módulos y Modularización**
- Eventos y callbacks
- Event loop y asincronía

npm

Es el gestor de paquetes de Node.js, nos permite descargar e instalar librerías y dependencias.
Cuando se instala Node, se genera un comando **npm** para usar en la terminal. Se debe inicializar utilizando el comando "**npm init**". Este comando creará un archivo **package.json**[1], dentro del cual se irán guardando todas las configuraciones y metadatos del proyecto.
La propiedad "**main**" (del package.json) hace referencia al **entry point** de nuestra aplicación, por convención el archivo principal lo llamamos "**index.js**". Si usamos el comando **"npm init -y"** inicializará un servidor por defecto y crea el package.json.
Para descargar las dependencias de un proyecto debemos escribir en la terminal "**npm install**".

[1] Dentro de la estructura del package.json podemos guardar en "scripts" algunos comandos personalizados para facilitar la ejecución de algunas tareas comunes. Si el nombre del comando no está reservado se debe agregar run.
comando: **npm run <nombre_del_script>**

Estructura del package.json

```
{
“name” : “nombre_de_la_carpetar”,
“version” : “1.0.0”,
“description” : “”,
“main” : “index.js”,
“scripts” : {
  “test” : “echo \ “Error: no test specified\”” && exit 1”
},
“author” : “”,
“license” : “ISC”,
“dependencies” : {
  “moment” : “^2.24.0” }
}
```

Instalando librerias

Para instalar una librería usamos el siguiente comando: **npm install PACKAGE --save** en donde reemplazamos la palabra **PACKAGE** por el nombre de la librería que queramos instalar. El comando **--save** guarda dentro del package.json, en la propiedad **“dependencies”**, una referencia a la librería que estamos instalando.

Dentro de la carpeta **node_modules**[2] se irán creando las carpetas de las librerías que instalemos.

Módulos y modularización (Node.js)

Tipos principales de módulos:

- **módulos nativos** | aquellos que ya vienen instalados como **fs** (file system), **http**, **path**, entre otros. Para requerir un módulo nativo usamos la función **require()** y le pasamos como argumento el nombre del módulo que requerimos.

- **módulos de terceros** | los que podemos instalar con NPM como express o mongoose.

```
npm install moment --save
const moment = require(‘moment’);
console.log(moment().format(‘MMMM Do YYYY, h:mm:ss a’));
```

- **módulos creados** | los definimos nosotros, con extensión js y exportamos con **module.exports**. Se importan con **require()**

```
module.exports = {series, peliculas}
```

Luego en el archivo principal, importamos “exportar.js” así:

```
const {series, peliculas} = require(‘./exportar’);
console.log(series);
```

Para exportar múltiples funciones o variables usamos llaves > module.exports = { … }

[2] Carpeta que se debe ignorar con .**gitignore**, ya que no es necesario que esté en nuestro proyecto por su tamaño. La información necesaria estará en el **package.json** y para eso usamos el comando **“npm install”** en la carpeta del proyecto y se instalarán las dependencias necesarias.

❌ Otra forma de exportar módulos es utilizando **export** e **import** (ES6 Modules).
Exportación nombrada (**export**), se puede declarar variables, funciones o clases y exportarlas directamente. Y luego importarlo >

```
import { f_sumar, f_restar } from './modulo.js'
```

Exportación por defecto (**export default**), puede exportar una única función, clase u objeto como la exportación principal del módulo, y en este caso no necesita las llaves.
Si voy a usar el **import** tengo que aclararlo en el package.json. > "**type" : "module**".

Introducción a HTTP

HTTP (HyperText Transfer Protocol) es el protocolo que gestiona las **transacciones web** entre clientes y servidores. Transferencia a través de **direcciones web** (técnicamente URI). Una **URI** (identificador de recursos uniformes) es un bloque de texto que se escribe en la barra de direcciones del navegador web y está compuesta por dos partes: la URL y la URN.

URL	URN
Indica dónde se encuentra el recurso y siempre comienza con un protocolo (HTTP)	Es el nombre exacto del recurso uniforme. El nombre del dominio o del recurso.
https://arbustra.net/en/home	

CÓDIGOS DE ESTADO HTTP
El código tiene tres dígitos. El primero representa uno de los 5 tipos de respuestas posibles:

1 _ _ Respuestas informativas
2 _ _ Respuestas exitosas
3 _ _ Redirecciones
4 _ _ Errores del cliente
5 _ _ Errores del servidor

HTTP en Node.js

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

```
const http = require('http')
http.createServer ( (req, res) => {
//cuerpo del callback
}).listen(3030, 'localhost');
```

createServer(), se encargará de levantar el servidor y manejar las peticiones que le lleguen. El método recibe como parámetro un **callback**, que se ejecutará cada vez que se envíe un request al servidor. El callback recibirá dos parámetros:

- **req**: representa los datos que envió el cliente como **solicitud**.
- **res**: representa la **respuesta** que le enviará el servidor al cliente.

listen(), método que define el puerto en el que el servidor escuchará las peticiones. Recibe dos parámetros:

- **3030**: el puerto donde se escuchará la app (cualquier número de 4 dígitos).
- **localhost**: el dominio donde queremos que se ejecute el servidor.

Levantar Servidor

Hacemos uso de la **consola** para ejecutar nuestro archivo entry-point. **node index.js**
Al ejecutar ese comando, la consola quedará inhabilitada. para cortar el servidor, presionamos **ctrl + c**.
Para testear el servidor le pedimos al navegador que le haga un request en **localhost:3030**.

Response (res)
Lo primero que hay que hacer es definir las cabeceras. Para crearlas usaremos el método **writeHead()** que lo ejecutamos sobre el parámetro **res**.

- primer parámetro: un número de tres dígitos (status de la petición).
- segundo parámetro: un objeto literal que define el tipo de contenido que se le está enviando al cliente.

También definimos el contenido que le enviaremos al cliente. Método **end()** que recibe como parámetro un string, que representa el **cuerpo** del contenido que estaremos enviando.

```
http.createServer ( (req, res) => {
res.writeHead ( 200, { "Content-Type" : "text/plain" } ) ;
res.end ( "cuerpo del contenido que estamos enviando al cliente" ) ;
}).listen(3030, 'localhost');
```

Express

Es un **framework** [3]que facilita y agiliza el desarrollo de aplicaciones web con Node.js.
Para usar express hay que instalar la librería en un proyecto Node ya iniciado, es decir haber hecho npm init y tener creado el archivo package.json. > **npm install express –save**

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 todas las propiedades y métodos de la librería disponibles.

```
const app= express();
```

SERVIDOR HTTP EN EXPRESS

```
const express = require('express');
const app= express();
app.listen(3030,()=> console.log("servidor corriendo"));
```

El método **listen** recibe dos parámetros: el primero, el número del puerto donde se ejecutará la aplicación y el segundo (opcional), un callback que retorna un console.log.

Ahora nos falta definir las rutas para empezar a manejar los response de nuestra app.
DEFINIENDO UNA RUTA

```
app[4].get[5]('/'[6],(req,res)[7] => {
   res.send('hola mundo'); // cuerpo del Handler
} )
```

---

[3] Un framework es un entorno de trabajo que trae resueltas una serie de tareas, automatizando así el desarrollo de cualquier aplicación.
[4] **app**: variable que guarda la ejecución de Express.
[5] **Metodo HTTP**: get, post, put, patch or delete.
[6] **Path**: string que hará referencia a la ruta en sí.
[7] Parámetros del Handler

Al objeto **app** le pedimos el método **get**.
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.

Dentro del callback definimos la respuesta que enviaremos < **res.send()** >. Al objeto **res**, le pedimos el método **send**. Como parámetro le pasamos lo que queremos mostrar en el browser, en este caso un texto. Otras respuestas:

- **res.sendFile()** | para enviar archivos HTML como respuesta.
- **res.json** | para enviar respuesta en formato JSON.

Rutas con parámetros
Se usa para manejar rutas dinámicas, donde se pueden recibir valores dentro de la URL que varían entre diferentes solicitudes. Esto permite responder de manera específica a cada solicitud sin necesidad de definir rutas estáticas para cada caso.

Cuando definimos una ruta con parámetros en Express, los especificamos en la URL con dos puntos seguidos del nombre del parámetro.

```
app.get('/user/:id', (req,res) => {
    const userId = req.params.id; // accedes al parámetro id
    res.send(`User ID: ${userId}`);
});
```

En este caso **:id** es un parámetro de la ruta. Esto significa que cualquier valor después de **/user/** será capturado como **id**. Al acceder a **/user/123**, por ejemplo la respuesta será: “User ID: 123”.

CLASE 16: Modularización

Modularización

Consiste en dividir el código en diferentes archivos y módulos.

- **index.js** | Archivo principal que configura el servidor, establece el puerto y define el middleware, como cors y express.json().
- **postControllers.js** | Contiene las funciones CRUD para los “posteos” y gestiona las solicitudes HTTP.
- **db.js** | Configura la conexión a la base de datos MySQL utilizando Sequelize.
- **postModel.js** | Define el modelo de datos postModel para la tabla “posteos” en la base de datos, usando Sequelize.
- **postRoutes.js** | Define las rutas de la API para los “posteos”

Middleware en Express

Son funciones que se ejecutan durante el ciclo de vida de una solicitud antes de enviar la respuesta. Se usa para procesar datos de solicitudes, verificar autenticaciones, registrar actividades, entre otras cosas.

- **express.json()** | Este middleware convierte el cuerpo de las solicitudes entrantes a formato JSON automáticamente.
- **cors** | Este middleware permite el “intercambio de recursos de origen cruzado”, lo que facilita las solicitudes desde diferentes dominios. Es útil para la comunicación entre el frontend y backend. **npm install cors**.

Sequelize

ORM para manejar la conexión y consultas a MySQL. La conexión se define mediante una nueva instancia de Sequelize con nombre de la tabla, usuario, contraseña, {host, dialecto y puerto}.
Instalar sequelize: **npm install sequelize mysql2**.

Nodemon

Reinicia el servidor automáticamente cada vez que se detectan cambios.
Instalar nodemon: **npm install -g nodemon**.

CLASE 17: Express Generator, MVC.

Express cuenta con un generador de proyectos llamado **express-generator**, que permite inicializar una aplicación con un esqueleto de carpetas, archivos y dependencias.

En la carpeta **raíz** encontraremos el entry point **app.js**, y el archivo **package.json**.

Dentro de la carpeta **bin** encontraremos el archivo **www** sin extensión. El mismo tiene 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** encontraremos dos vistas iniciales que trae el generador: **index.js** y **error.ejs**.

⚠️ 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.

Instalando express-generator

comando: `npm install -g express-generator`

Lo próximo será crear un proyecto Node usando Express con un comando que creará la carpeta del proyecto con el nombre que definamos nosotros. Usaremos EJS
comando: `express nombre_proyecto –ejs`

Por último, dentro de la carpeta del proyecto, tendremos que correr el comando para instalar todas las dependencias.
comando:
```
cd nombre_proyecto
npm install
```

Levantar servidor

Dentro de la carpeta del proyecto debemos iniciar el servidor con nodemon.
comando: `nodemon bin/www`

Por último ingresar a http://localhost:3000

Motor de vista

Los motores de vistas, también conocidos como motores de plantilla o template engines, nos permiten crear una estructura dinámica para las vistas de nuestro proyecto.
Entre los más nombrados se encuentran **EJS**, **Underscore**, **Handlebars**, entre otros.

Introducción a MVC

Es un patrón de diseño MVC (Modelo Vista Controlador). Su objetivo es crear aplicaciones modulares, dividiendo el proyecto en tres componentes principales. Estos componentes son: los modelos, las vistas y los controladores.

Las Vistas

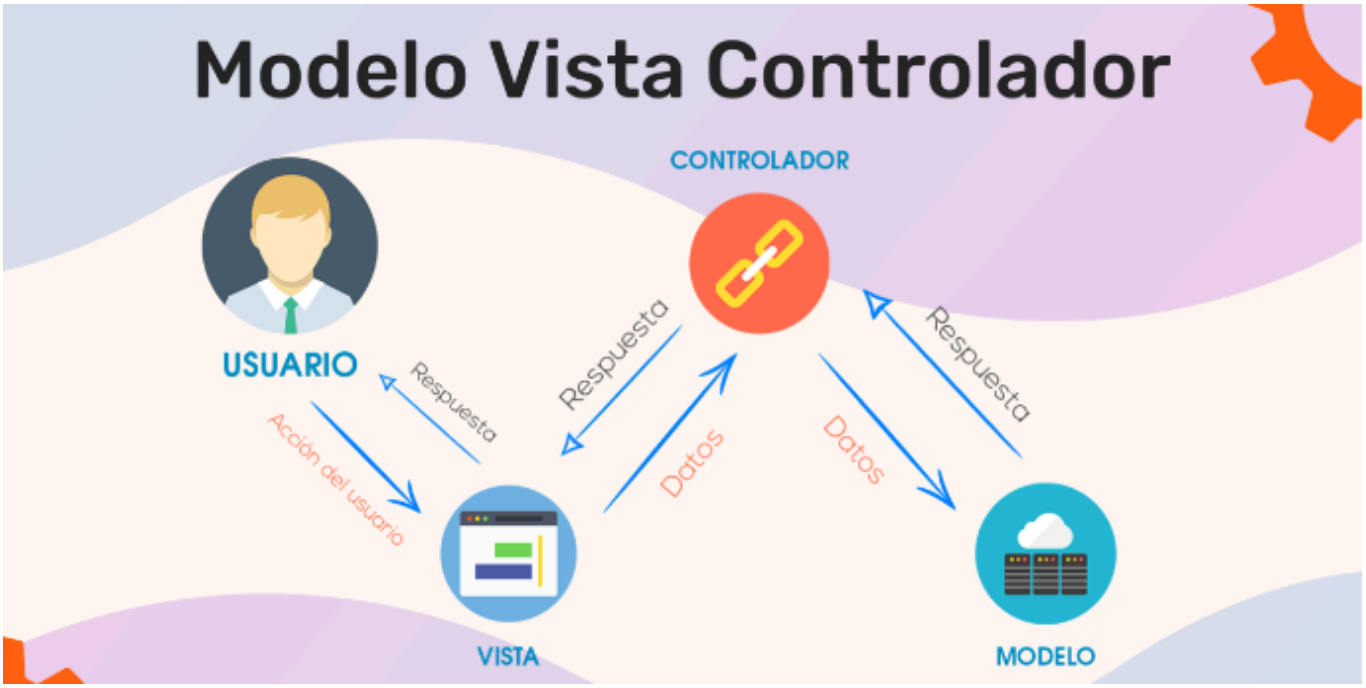
Conforman la interfaz gráfica de la aplicación y contienen todos los elementos que son visibles al usuario. A través de ellas el usuario interactúa enviando y solicitando información al servidor. Su responsabilidad es definir la apariencia de datos y mostrarlos en pantalla. Las vistas no se comunican de forma directa con los modelos.

Los Modelos

Conforman y contienen la lógica de la aplicación. Su responsabilidad es 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 elegir 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.



Ejemplo de 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.
El **controlador** recibe la petición y le solicita al **modelo** el detalle del producto 20.
El **modelo** busca la información solicitada y se la envía al **controlador**.
El **controlador** recibe la información y le envía los datos a la **vista**.
La **vista** le muestra al usuario los datos que recibió.

Renderizar una vista

Para poder renderizar una vista es importante aclararle al controlador qué vista queremos enviar al navegador.

.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.

⚠️ No hace falta aclararle la **carpeta** en donde está almacenada esa vista <siempre y cuando hayamos configurado el template engine correctamente con método use()>.
⚠️ Tampoco hace falta aclarar la **extensión** del archivo.

```
cont controller = {
   mostrarPeliculas:(req,res) => {
      res.render('peliculas')
   }
}
```

Parámetros Compartidos

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 a renderizar.

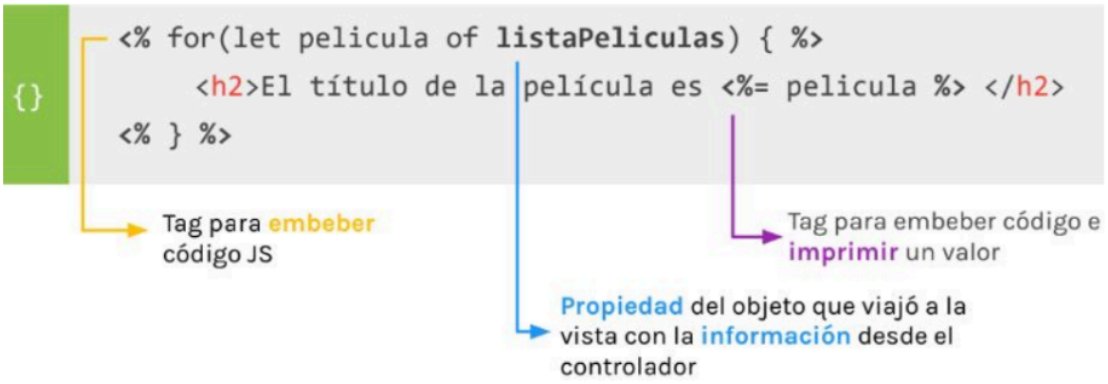
```
cont peliculas = ['Deadpool','the Joker','Batman'];

cont controller = {
        mostrarPeliculas:(req,res) => {
           res.render('peliculas',{listaPeliculas: peliculas,
                                   genero: 'superheroes'})
        }
     }
```

El nombre de la **propiedad** (**listaPeliculas**) será el que usaremos para disponer de esa información dentro del archivo de la vista.
El **valor** (peliculas) será la información que queremos que viaje hasta la vista.

La vista, por su parte, recibirá las propiedades sueltas como variables.

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.



Tags de EJS

EJS trae consigo un conjunto de etiquetas que nos permiten integrar funcionalidad de js dentro de nuestras estructuras html.

Estructura Básica | **<% … %>**
Esta etiqueta nos permite **incorporar** código de JavaScript, por cada línea de js que escribamos, debemos encerrar ese código con la etiqueta **<%...%>**.
También nos permite **imprimir** un valor dinámico (el resultado) usando **<%= … %>**. En este caso el HTML será escapado.
Usamos **<%- … %>** para renderizar contenido **sin escapar** el HTML.

Recursos Estáticos

Son aquellos recursos públicos que manejamos dentro de nuestra aplicación: imágenes, archivos css, archivos js, etc.
Para poder disponer libremente de ellos en nuestro proyecto, hace falta aclararle a Express dónde vamos a estar almacenando esos recursos.
Con la siguiente línea de código le estaremos dando a Express acceso libre a todo lo que se encuentre dentro de la carpeta public.

```
app.use(express.static(__dirname + '/public');
```

Para requerir un recurso estático, solo hace falta aclarar la ruta hacia dicho recurso, comenzando la ruta siempre con una barra.

Paso a paso para configurar EJS y recursos estáticos

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'));
```