

Manual de Diseño Web con HTML5, CSS3 y JavaScript

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36



HTML

Módulo I –

Introducción

Página oficial en MDN

<https://developer.mozilla.org/es/docs/Web/HTML>

El siguiente manual es una guía práctica de los contenidos del módulo que sirve como una guía de consulta rápida y guía de estudio, pero nunca como sustituto a la teoría/práctica de las clases o a la documentación oficial.

Recuerda estudiar siempre desde la documentación oficial, por eso añadimos los enlaces oficiales en cada sección.

HTML (*Hypertext Markup Language*) es el lenguaje de marcado estándar utilizado para la creación de páginas web. Permite describir la estructura de un documento web, mediante etiquetas que especifican elementos tales como encabezados, párrafos, enlaces, imágenes, entre otros.

En su definición más simple, hipertexto es un texto que contiene enlaces a otros textos, sin embargo, en la actualidad, la naturaleza de esos textos es completamente multimedia, siendo una especie de contenidos relacionados entre sí que pueden consumirse prácticamente en cualquier orden. Un lenguaje de etiquetas nos permite definir o darle significado a partes de un texto utilizando unas secuencias de caracteres predefinidos (las etiquetas).

Utilizando HTML vamos a definir/representar la estructura y el contenido de un documento web mediante etiquetas.



HTML

Módulo II –

Etiquetas y atributos

HTML utiliza etiquetas para definir los diferentes elementos en una página web, como imágenes, texto y enlaces. Cada etiqueta puede contener varios atributos, que proporcionan información adicional o especifican características del elemento.

Es habitual pensar que si la web se visualiza bien y ese elemento que has añadido aparece bien ya está bien lo que hemos escrito pero comprender cómo funcionan conjuntamente las etiquetas y atributos es crucial para crear documentos HTML, si la semántica del documento HTML no está bien relacionada HTML no tiene sentido.

Etiquetas

Página oficial en MDN

[https://developer.mozilla.org/es/docs/
Web/HTML/Element](https://developer.mozilla.org/es/docs/Web/HTML/Element)

Una etiqueta nos permite delimitar o hacer referencia a un elemento, ya sea semántico o estructural. Están delimitadas por los símbolos de menor y mayor que: < y >, y entre ellas está una palabra o le-

tra que representa el tipo de elemento, por ejemplo: <p> representa un párrafo (*paragraph*).

Existen varios tipos de etiquetas:

- Etiquetas de apertura <elemento> que representan el punto en donde inicia un elemento, etiquetas de cierre </elemento> que representan el punto en donde termina un elemento.
- Etiquetas (vacías) que se cierran a sí mismas <elemento /> las cuales se utilizan para elementos que hacen referencia a entidades no textuales, como imágenes, entre otras.

Atributos

Página oficial en MDN

<https://developer.mozilla.org/es/docs/Web/HTML/Attributes>

Un atributo define una característica de un elemento; siempre se encuentran en la etiqueta de apertura y están compuestos de un nombre, un símbolo de igualdad y el valor del atributo entre comillas.



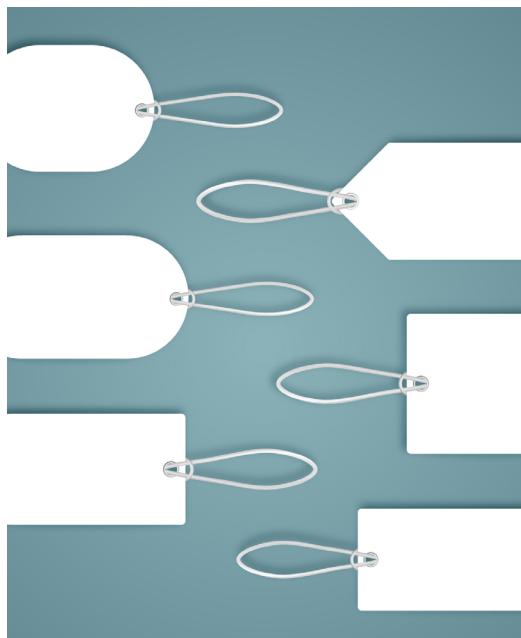
```
<etiqueta atributo="valor">CONTENIDO</etiqueta>
<etiqueta atributo="valor" />

<!-- este tipo en particular de elementos representan comentarios, textos que
el lector de HTML ignora y que sirven para documentar nuestro código --&gt;</pre>
```

La estructura básica de un documento HTML es la siguiente:



```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    CONTENIDO DE LA WEB (Cabecera, párrafos, pié de página ... )
  </body>
</html>
```



HTML

Módulo III –

Etiquetas de contenido

Página oficial en MDN

[https://developer.mozilla.org/es/docs/
Web/HTML/Element](https://developer.mozilla.org/es/docs/Web/HTML/Element)

Las etiquetas de contenido en HTML son fundamentales para estructurar y presentar la información en una página web. Desde texto y enlaces hasta imágenes y listas, estas etiquetas permiten organizar el contenido de manera coherente y accesible.

Encabezados (*headings*)

Página oficial en MDN

[https://developer.mozilla.org/en-US/
docs/Web/HTML/Element/Heading_Ele-
ments](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/Heading_Elements)

Los encabezados en HTML, que van desde `<h1>` hasta `<h6>`, definen la jerarquía del contenido en una página. `<h1>` se utiliza para el título principal, mientras que `<h2>` a `<h6>` se utilizan para subtítulos y encabezados secundarios, decreciendo

en importancia y generalmente en tamaño. El uso adecuado de los encabezados facilita una estructura clara y accesible.



```
<h1>Título principal</h1>
<h2>Título secundario</h2>
<h3>etc.</h3> <h4>etc.</h4>
<h5>etc.</h5>
<h6> ... </h6>
```

Una confusión común es pensar que los números representan tamaño (fácil de caer en ella por la manera nativa de representarlos del navegador), sin embargo representan jerarquías: los h2 titularán secciones de algo titulado con h1, los h3 titularán secciones dentro de las secciones tituladas con h2, y así sucesivamente.

Párrafos

La etiqueta **<p>** define bloques de texto en forma de párrafos. Esta etiqueta ayuda a separar y organizar el contenido textual en segmentos legibles y coherentes. Además de mejorar la legibilidad, el uso correcto de **<p>** es fundamental para garantizar una estructura lógica y una experiencia de usuario óptima.

Un error habitual es añadir un **<p>** con una palabra, o frases de tres palabras, que no deberían estar en un **<p>**.



```
<p>Esto sería un párrafo lorem ipsum dolor sit amet consectetur adipisicing elit. Repellendus voluptate eligendi rem quibusdam accusamus minima quas et natus aliquam adipisci!</p>
```

```
<p>Y esto, otro párrafo lorem ipsum dolor sit amet consectetur adipisicing elit. Repellendus voluptate eligendi rem quibusdam accusamus minima quas et natus aliquam adipisci!</p>
```

Hipervínculos

La etiqueta **<a>**, o etiqueta de anclaje, define hipervínculos en HTML. Los hipervínculos permiten a los usuarios navegar entre diferentes páginas web o recursos, creando una red interconectada de información. También puede contener atributos como **href**, que especifica la URL del recurso vinculado o **title** que define una descripción del propio enlace.



```
<a href="url/del/recurso" title="Descripción">texto del enlace</a>
```

Imágenes

La etiqueta **** incorpora imágenes en una página web y utiliza el atributo **src** para especificar la dirección de la imagen. El atributo **alt** proporciona una descripción textual de la imagen, mostrada si la imagen no puede cargarse y útil para la accesibilidad.

Para optimizar la presentación de las imágenes, se pueden usar atributos como **width** y **height** para definir las dimensiones.

Es importante considerar aspectos como la resolución y el formato de archivo para asegurar una carga rápida y una visualización adecuada en diferentes dispositivos.

El atributo **src** (*source*) le indica al navegador la URL (*Uniform Resource Locator*) de la imagen, y puede ser de dos tipos: relativa y absoluta.



```
!— Teoría de una imagen —>

```

Tablas

La etiqueta **<table>** crea tablas en HTML, estructurando datos en filas y columnas. Una tabla se compone de filas (**<tr>**), y cada fila contiene celdas (**<td>** o **<th>**) para datos y encabezados de columna respectivamente. Las tablas son herramientas para la presentación de datos de manera ordenada y coherente.



```
<table><!-- elemento estructural --&gt;
  &lt;thead&gt;<!-- elemento estructural, cabecera de la tabla --&gt;
    &lt;tr&gt;<!-- elemento estructural fila de tabla (table row) --&gt;
      &lt;td&gt;El elemento td representa el contenido, una celda&lt;/td&gt;
      &lt;td&gt;Representan las columnas&lt;/td&gt;
      &lt;td&gt;de una fila (tr)&lt;/td&gt;
    &lt;/tr&gt;
  &lt;/thead&gt;
  &lt;tbody&gt;<!-- elemento estructural, cuerpo de la tabla, registros ... --&gt;
    &lt;tr&gt;<!-- elemento estructural fila de tabla (table row) --&gt;
      &lt;td&gt;celdas ... &lt;/td&gt;
    &lt;/tr&gt;
  &lt;/tbody&gt;
  &lt;tfoot&gt;<!-- elemento estructural, pie de la tabla, registros ... --&gt;
    &lt;tr&gt;<!-- elemento estructural fila de tabla (table row) --&gt;
      &lt;td&gt;celdas ... &lt;/td&gt;
    &lt;/tr&gt;
  &lt;/tfoot&gt;
&lt;/table&gt;</pre>
```



HTML

Módulo IV –

Etiquetas estructurales

Página oficial en MDN

<https://developer.mozilla.org/es/docs/Web/HTML/Element>

Elementos estructurales

Existen dos elementos estructurales en HTML, los cuales se utilizan para agrupar, o en algunos casos separar, unos elementos de otros:

- **div** son elementos que permiten delimitar, agrupar o contener otros elementos, son elementos de tipo *block*.
- **span** son elementos que permiten delimitar, agrupar o contener letras o palabras, son elementos de tipo *inline*.

Elementos estructurales-semánticos

Existe otra serie de elementos que nos permiten delimitar, agrupar o contener otros elementos, pero que a diferencia de los **div**, cumplen una función semántica específica, los más importantes:

- **header** contiene todos los elementos que forman la cabecera del documento.

- **footer** contiene todos los elementos que forman el pie del documento.
- **nav** contiene todos los elementos que forman la navegación de la web.
- **section** contiene todos los elementos que forman una sección del documento, su uso depende en gran medida del contenido.
- **article** contiene todos los elementos que forman un artículo en el documento, su uso depende en gran medida del contenido.

Además de **header**, **footer**, **nav**, **section** y **article** existen otras etiquetas semánticas como **aside**, que se utiliza para contenido tangencial o secundario, y **main**, para especificar el contenido principal del documento. Otras etiquetas incluyen **figure** para referenciar contenido multimedia y **figcaption** para proporcionar una leyenda a dichos contenidos.

Listas

Las listas son elementos estructurales que por dentro llevan elementos de contenido, existen varios tipos de listas aunque habitualmente se usan las desordenadas y las ordenadas.

- Listas no ordenadas
- Listas ordenadas
- Listas de definiciones

```
● ● ●

<!-- Lista desordenada -->
<ul>
  <li>item de lista (list item), representa un ítem (contenido)</li>
  <li>Se necesita un li por cada ítem individual</li>
</ul>

<!-- Lista ordenada -->
<ol>
  <li>item de lista (list item), representa un ítem (contenido)</li>
  <li>Se necesita un li por cada ítem individual</li>
</ol>

<!-- Lista definición -->
<dl>
  <dt>Término</dt>
  <dd>Definición</dd>
  <dd>Definición</dd>
</dl>
```



HTML

Módulo V –

Formularios

Página oficial en MDN

<https://developer.mozilla.org/es/docs/Learn/Forms>

Representan un formulario a enviar por el usuario, el mismo debe contener campos a llenar por el usuario.

Debe tener dos atributos: **method** y **action**; **method** define el método HTTP a utilizar en el momento del envío (GET o POST) y **action** representa a dónde enviará (URL) los datos el formulario.



```
<form method="POST" action="url/de/envío" encytype="multipart/form-data">
  <input>!-- estos elementos serán explicados en el próximo apartado -->
</form>
```

Inputs

Página oficial en MDN

<https://developer.mozilla.org/es/docs/Web/HTML/Element/input>

Los elementos **input** pueden tener diversos atributos que controlan su comportamiento y presentación. Además de **name**, **value**, y **placeholder**, atributos como **rea-**

donly y **disabled** controlan la interacción del usuario. **readonly** hace que un campo de entrada sea de solo lectura, mientras que **disabled** desactiva el campo. Otros atributos incluyen **maxlength**, que limita la cantidad de caracteres permitidos, y **required**, que indica que el campo debe ser llenado antes de enviar el formulario.

Pueden tener otros atributos opcionales, entre los más importantes:

- **name** define el nombre del *input*.
- **value** define el valor predeterminado del *input*.
- **placeholder** define un texto sustitutivo para guiar al usuario.

A continuación, los tipos de *input* más importantes con sus atributos relacionados (en donde aplique):

```
● ● ●

<input type="text">      ←!—Texto →
<input type="password"> ←!—Contraseña →
<input type="number" min="0" max="20">←!— Números →
←!— Selección única, si varios comparten el mismo nombre,
    se comportan como un solo input con múltiples opciones,
    de las cuales solo una puede estar seleccionada,
    el valor de dicho input será el del seleccionado,
    el atributo checked (boolean) indica que estará seleccionado por defecto
→
<input type="radio" name="x" value="y" checked>

←!—Selección múltiple, si varios comparten el mismo nombre,
    se comportan como un solo input con múltiples opciones,
    de las cuales se pueden seleccionar varias,
    el atributo checked (boolean) indica que estará seleccionado por defecto
→
<input type="checkbox" name="x" value="y" checked>
```

Otras etiquetas de formulario

También existen otras etiquetas como:

- **select** permite seleccionar a partir de una lista de opciones
- **textarea** permite escribir un texto de longitud superior a un simple campo de tipo *text*

Además de **select** y **textarea**, existen otras etiquetas relacionadas con formularios. La etiqueta **button** define botones clicables, mientras que **label** proporciona una etiqueta textual para un elemento de formulario.

```
● ● ●

<!-- Elemento estructural que contiene las opciones -->
<select name="ciudad">
    <!--Se puede utilizar una opción pre seleccionada e inactiva como
placeholder -->
    <option selected disabled>seleccione un país</option>
    <option value="1">Cádiz</option>
    <option value="2">Madrid</option>
    <option value="3">Sevilla</option>
</select>

<textarea name="descripcion" rows="10" cols="40">
    Se puede escribir un placeholder directamente entre las etiquetas,
    los atributos rows y cols definen la cantidad de filas (líneas) y
    columnas ("caracteres") que se pueden escribir.
</textarea>
```



CSS

Módulo VI –

Introducción

Introducción

Página oficial en MDN

[https://developer.mozilla.org/es/docs/
Web/CSS](https://developer.mozilla.org/es/docs/Web/CSS)

CSS, que significa *Cascading Style Sheets* (Hojas de Estilo en Cascada), es un lenguaje de diseño que se utiliza para describir la presentación de documentos escritos en HTML o XML. Al separar el contenido del diseño, CSS permite a los desarrolladores controlar aspectos visuales como el color, la disposición y el tamaño de los elementos en una página web, mejorando la accesibilidad y permitiendo una experiencia de usuario más rica y coherente.

Este lenguaje es una parte integral del desarrollo web y facilita la creación de interfaces de usuario visualmente atractivas y responsivas. CSS se ha expandido y evolucionado a lo largo del tiempo, con varias especificaciones y módulos que se añaden para ofrecer mayor control y flexibilidad en el diseño web.

Selector

Página oficial en MDN

https://developer.mozilla.org/es/docs/Web/CSS/CSS_selectors

Los selectores en CSS son patrones que se utilizan para seleccionar y manipular elementos HTML.

Pueden hacer referencia a tipos de etiquetas específicas, identificadores, clases, atributos o estados de un elemento. Al determinar el alcance de la aplicación de una regla de estilo, los selectores ofrecen gran precisión y flexibilidad en el diseño.

Propiedad

Una propiedad en CSS define qué aspecto del elemento HTML se va a estilizar. Cada propiedad tiene un nombre y está asociada con un valor específico. Por ejemplo, la propiedad **color** controla el color del texto, mientras que **font-size** determina el tamaño de la fuente. Las propiedades y sus valores permiten a los desarrolladores controlar detalladamente el estilo de los elementos en una página.

Regla

Una regla en CSS se compone de un selector y un bloque de declaración. El bloque de declaración contiene una o más declaraciones, cada una compuesta por una propiedad y un valor. Por ejemplo, la regla **p {color: blue;}** especifica que todos los párrafos (**<p>**) deben tener un color de texto azul. Las reglas permiten aplicar estilos de manera coherente y modular.

```
/* Regla */

p{ /* Selector */
    color: blue; /* nombre-propiedad y su valor; */
}
```

Usando CSS

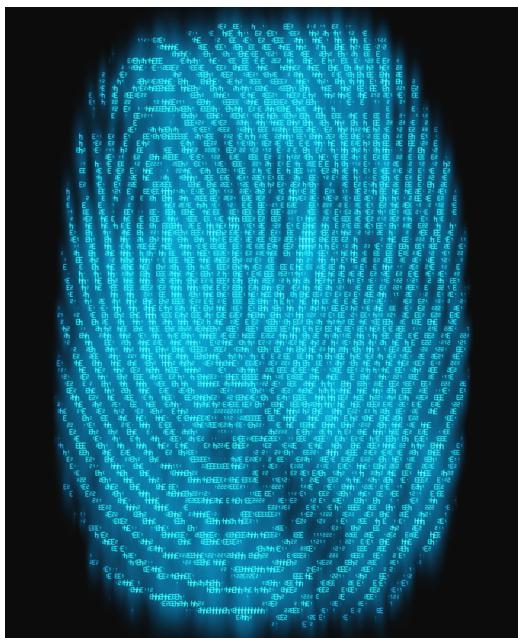
Integrar CSS con HTML es crucial para diseñar y estilizar páginas web de manera efectiva. Hay diferentes métodos para incluir estilos CSS en documentos HTML, y cada uno de ellos tiene sus propias ventajas y casos de uso.

Formas de usar CSS en HTML:

1. **Eiqueta link:** utiliza la etiqueta `<link>` en el `<head>` del documento HTML para referenciar un archivo externo de CSS. Este método es recomendable para mantener los estilos separados del contenido, facilitando la mantenibilidad del código.
2. **Eiqueta style:** este método incluye el código CSS directamente dentro del documento HTML utilizando la etiqueta `<style>`. Resulta útil para estilos específicos de una página en particular.
3. **Atributo style:** se pueden aplicar estilos directamente a un elemento HTML específico usando el atributo `style`. Esto se utiliza generalmente para estilos únicos y específicos que no se repetirán en otros lugares.



```
!— Ejemplo de cómo usar CSS junto a HTML —  
!— 1. Usando la etiqueta link —>  
<head>  
  <link rel="stylesheet" href="styles.css">  
</head>  
  
!— 2. Usando la etiqueta style —>  
<style>  
  p {  
    color: blue;  
  }  
</style>  
  
!— 3. Usando el atributo style —>  
<p style="color:red;">Este es un párrafo con texto rojo.</p>
```



CSS

Módulo VII–

Selectores

Página oficial en MDN

https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/Selectors

Los selectores en CSS son fundamentales para aplicar estilos a distintos elementos HTML de una página web.

Al funcionar como un criterio de búsqueda, permiten identificar elementos específicos en el árbol del documento y, por ende, determinar a qué elementos se les aplicará un conjunto de reglas de estilo.

Los selectores pueden ser desde simples, como seleccionar por tipo de etiqueta, hasta complejos, como seleccionar el primer hijo de un elemento.

Existen diversos tipos de selectores, cada uno con su propio nivel de especificidad y funcionalidad.

Especificidad

La especificidad en CSS se refiere a la prioridad que tiene una regla de estilo en relación con otras. Cuando múltiples reglas se aplican a un mismo elemento, la especificidad determina cuál de ellas prevalecerá.

Esta se calcula según una jerarquía: los selectores de ID tienen una especificidad mayor que los selectores de clase, atributo y tipo, que a su vez tienen una especificidad mayor que los selectores universales.

Selectores básicos

Los selectores básicos son los pilares para empezar a aplicar estilos en CSS. Permiten seleccionar elementos basándose en su ID, clase, tipo de etiqueta o de forma universal.

- **Selector de ID:** selecciona un elemento basado en su atributo **id**. Tiene alta especificidad (0,1,0,0).
- **Selector de Clase:** selecciona elementos basados en su atributo **class**. Su especificidad es media (0,0,1,0).
- **Selector de Etiqueta:** selecciona elementos basados en su tipo de etiqueta. Tiene baja especificidad (0,0,0,1).
- **Selector Universal:** selecciona todos los elementos. Tiene la especificidad más baja (0,0,0,0).

```
● ● ●  
#id{}    /* Selector de ID */  
.clase{} /* Selector de Clase */  
p {}      /* Selector de Etiqueta */  
* {}      /* Selector Universal */
```

Pseudo-selectores

Los pseudoselectores en CSS permiten seleccionar elementos basándose en su estado o posición relativa en el documento. Ofrecen una gran versatilidad al permitir aplicar estilos sin modificar la estructura HTML.

- **:hover:** selecciona un elemento cuando el cursor está sobre él.
- **:first-child:** selecciona el primer hijo de un elemento.
- **:last-child:** selecciona el último hijo de un elemento.

```
● ● ●  
a:hover { } /* Aplica estilos al pasar el cursor */  
li:first-child { } /* Selecciona el primer ítem de una lista */  
p:last-child { } /* Selecciona el último párrafo */  
input:checked { } /* Selecciona un checkbox o radio seleccionado */  
p:nth-child(2) { } /* Selecciona el segundo párrafo */
```

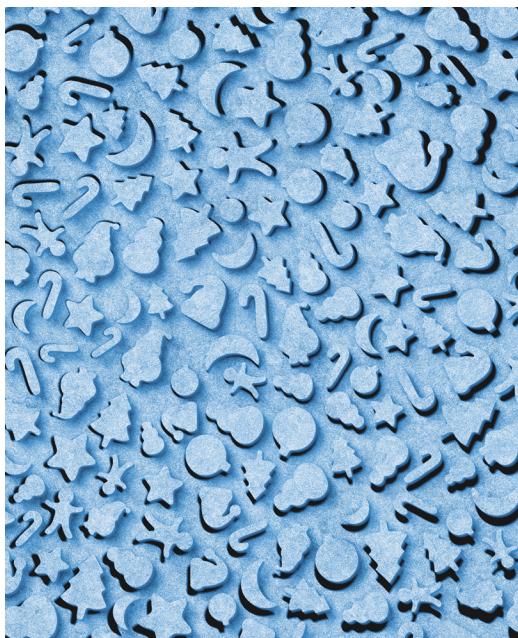
Pseudo-elementos

Los pseudo-elementos en CSS permiten estilizar partes específicas de un elemento, como su primera línea o su primer carácter, que no serían seleccionables de otra manera.

- **::before:** inserta contenido antes del contenido de un elemento.
- **::after:** inserta contenido después del contenido de un elemento.
- **::first-line:** selecciona la primera línea de un bloque de texto.



```
p::before { } /* Inserta contenido al inicio del párrafo */
p::after { } /* Inserta contenido al final del párrafo */
p::first-line { } /* Selecciona la primera línea del párrafo */
blockquote::first-letter{} /* Selecciona la primera letra de un blockquote */
div::selection { } /* Selecciona el texto seleccionado en un div */
```



CSS

Módulo VIII –

Unidades de medida

Página oficial en MDN

https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/Values_and_units

Las unidades de medida en CSS son esenciales para establecer valores precisos para diversas propiedades. Permiten que los desarrolladores definan tamaños, espacios, márgenes, y otros aspectos visuales de los elementos de una página web de manera consistente y adaptable a diferentes tamaños de pantalla y resoluciones.

```
div {  
    width: 50%;  
    font-size: 2em;  
    padding: 1rem;  
    margin-top: 10vw;  
    height: 80vh;  
}
```

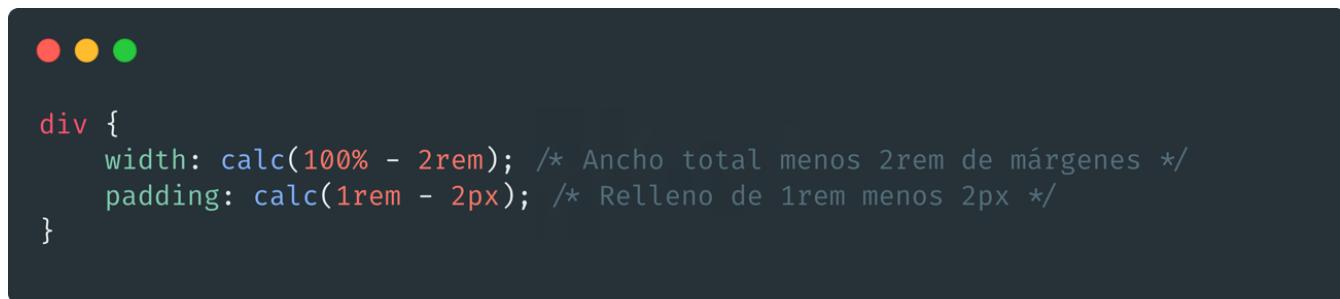
Cuando hablamos de medidas en CSS y tenemos en cuenta las principales, observamos como a nivel de accesibilidad la mejor de ellas es la medida **em** como medida general para cualquier propiedad.

En la vida real el «em» puede darnos varios problemas, como recalcular *margin* o *padding* si configuramos el *font-size* de un elemento, por ello si no utilizamos *em* para todo podemos definir que:

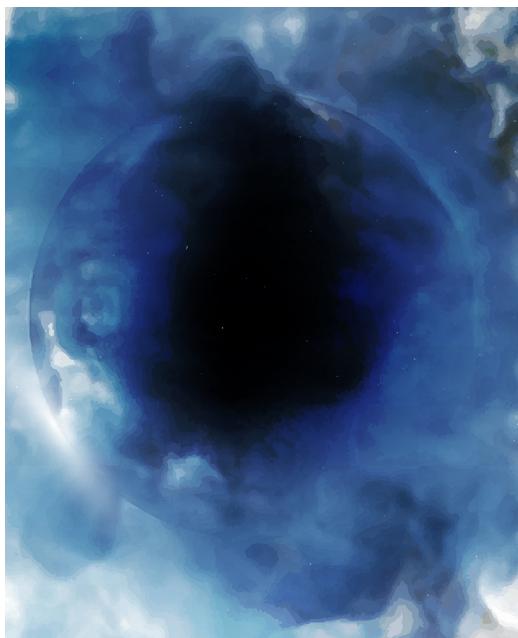
- **em** es la medida ideal para tipografías (*font-size*)
- **rem** es la medida ideal para propiedades box-model (*margin*, *padding*, *gap*, *border*...)
- **%** es la medida ideal para anchos (*width*) y alturas (*height*)
- **px** es la medida ideal para valores muy pequeños como 1px, 2px
- **fr** es la ideal para configurar la propiedad *grid-template-columns* con **display:grid**

calc

La función **calc()** en CSS permite realizar cálculos directamente en las hojas de estilo para determinar el valor de una propiedad. Es útil cuando se desean combinar diferentes unidades de medida o realizar cálculos entre ellas. Es importante tener en cuenta que debe haber un espacio antes y después del operador utilizado en **calc()**, de lo contrario, el valor de la propiedad se considerará inválido.



```
div {  
    width: calc(100% - 2rem); /* Ancho total menos 2rem de márgenes */  
    padding: calc(1rem - 2px); /* Relleno de 1rem menos 2px */  
}
```



CSS

Módulo IX-

Reset CSS/ Normalizadores

El uso de *Reset CSS* o normalizadores es esencial para crear una experiencia de usuario coherente en diferentes navegadores y plataformas. Los estilos predeterminados de los navegadores pueden variar significativamente, y usar un conjunto de reglas de reseteo ayuda a establecer un punto de partida común, eliminando inconsistencias y permitiendo que los diseñadores web tengan un mayor control sobre la apariencia de los elementos.

Lista de algunos *resets*:

1. **Reset de Meyer:** es uno de los *resets* más conocidos, creado por Eric Meyer. Este *reset* aborda una amplia gama de elementos y propiedades.
2. **Normalize.css:** a diferencia de un *reset* tradicional, **normalize.css** no elimina todos los estilos predeterminados sino que los hace más consistentes entre los navegadores.
3. **Reset CSS PRO:** es un *reset* que homogeniza los estilos de las cajas, tipografías y evita problemas en formularios.
4. **MiniReset.css:** es una versión más ligera y minimalista que solo resetea los estilos esenciales.

5. **Bootstrap Reboot:** Bootstrap incluye su propio conjunto de reglas de reseteo llamado ***Reboot***, que se basa en ***normalize.css*** pero con algunos ajustes adicionales.

```
html, body, div, h1, h2, h3, ul, ol, li, p, img {  
    margin: 0;  
    padding: 0;  
    border: 0;  
}
```



CSS

Módulo X-

Propiedades

Box-model

Página oficial en MDN

https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/The_box_model

Ancho y alto

El ancho (*width*) y el alto (*height*) de un elemento HTML son propiedades fundamentales en CSS que permiten controlar las dimensiones de dicho elemento. Estas propiedades pueden tener valores fijos (como **px**, **em**, **rem**) o porcentuales (%) en relación con el contenedor padre.

- **width:** define el ancho de un elemento.
- **height:** define el alto de un elemento.
- **max-width:** establece el ancho máximo de un elemento.
- **min-width:** establece el ancho mínimo de un elemento.
- **max-height:** establece la altura máxima de un elemento.
- **min-height:** establece la altura mínima de un elemento.



```
/* Ejemplo de Ancho y Alto */
div {
    width: 200px;
    height: 100%;
    max-width: 500px;
    min-height: 50px;
}
```

Margin

La propiedad **margin** en CSS se utiliza para crear espacio alrededor de los elementos, fuera de los bordes definidos. Es una manera efectiva de ajustar la distribución y alineación de los elementos dentro de un diseño.

- **margin-top:** establece el margen superior.
- **margin-right:** establece el margen derecho.
- **margin-bottom:** establece el margen inferior.
- **margin-left:** establece el margen izquierdo.



```
/* Ejemplo de Margin */
div {
    margin: 10px 15px 20px 25px; /* top, right, bottom, left */
    margin: 10px 15px 20px; /* top, right+left, bottom */
    margin: 10px 15px; /* top+bottom, right+left */
    margin: 10px; /* todos */
}
```

Padding

La propiedad **padding** se utiliza para generar espacio entre el contenido de un elemento y su borde. A diferencia de **margin**, **padding** agrega espacio dentro del elemento.

- **padding-top:** define el relleno superior.
- **padding-right:** define el relleno derecho.
- **padding-bottom:** define el relleno inferior.
- **padding-left:** define el relleno izquierdo.



```
/* Ejemplo de Padding */
div {
    padding: 10px 15px 20px 25px; /* top, right, bottom, left */
    padding: 10px 15px 20px; /* top, right+left, bottom */
    padding: 10px 15px; /* top+bottom, right+left */
    padding: 10px; /* todos */
}
```

Border

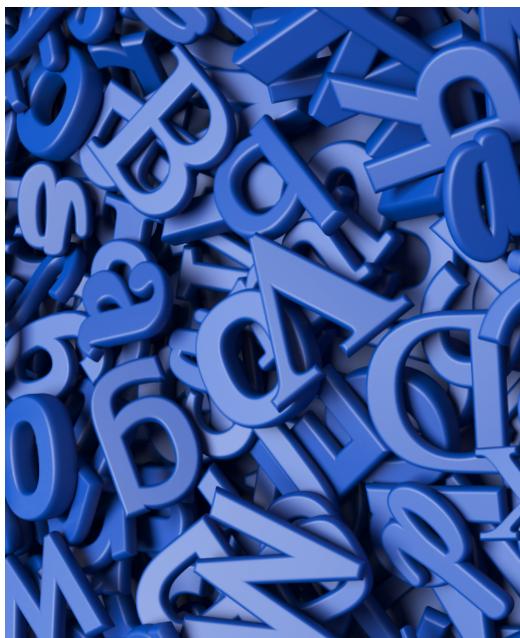
La propiedad **border** en CSS permite especificar el estilo, el ancho y el color del borde de un elemento.

Estas propiedades de **border** se especifican generalmente en un orden determinado: primero el ancho, luego el estilo y finalmente el color.

El borde de un elemento no solo proporciona una delimitación visual, sino que también puede influir en el espaciado y el diseño, especialmente cuando se combinan con **margin** y **padding**.



```
div {
    border: 2px solid #000000; /* Ancho, Estilo, Color */
}
```



CSS

Módulo XI-

Tipografía

Página oficial en MDN

<https://developer.mozilla.org/en-US/docs/Web/CSS/font>

Font

La tipografía es un elemento crucial en el diseño web y la propiedad **font** en CSS permite a los desarrolladores controlar el estilo, tamaño y familia de la fuente utilizada en un sitio web. Esta propiedad es, en realidad, un atajo que permite establecer varias propiedades relacionadas con la fuente en una sola línea.

- **font-family:** define la familia de la fuente a utilizar.
- **font-size:** establece el tamaño de la fuente.
- **font-weight:** controla el grosor de los caracteres.
- **font-style:** determina el estilo de la fuente, como cursiva o normal.
- **font-variant:** controla la variante de la fuente, como pequeñas mayúsculas.
- **line-height:** establece la altura de línea.

- **font-stretch:** permite ajustar el ancho de la fuente.

```
● ● ●

p {
    font-family: 'Arial', sans-serif;
    font-size: 16px;
    font-weight: bold;
    font-style: italic;
    font-variant: small-caps;
    line-height: 1.5;
    font-stretch: condensed;
}
```

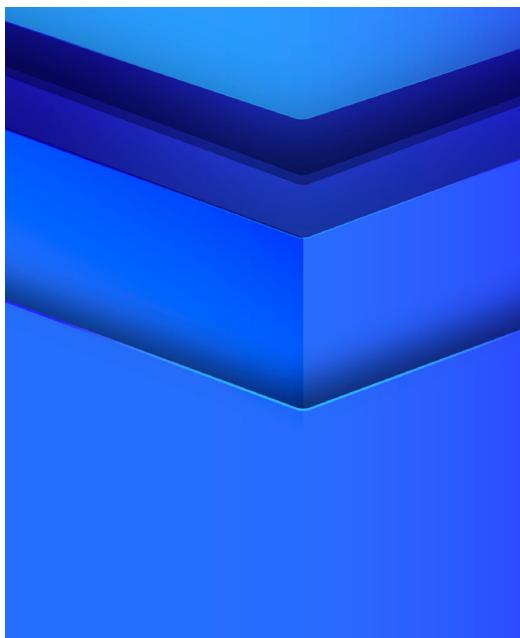
Text

Las propiedades de texto en CSS proporcionan control sobre la presentación del contenido textual en la web, permitiendo ajustar aspectos como la alineación, el espacio entre letras y el color del texto.

- **text-align:** establece la alineación horizontal del texto.
- **text-decoration:** aplica decoración al texto como subrayado, línea sobre el texto, etc.
- **text-indent:** establece el sangrado de la primera línea de un bloque de texto.
- **text-transform:** controla la capitalización del texto.
- **letter-spacing:** ajusta el espacio entre caracteres.
- **word-spacing:** ajusta el espacio entre palabras.
- **text-shadow:** aplica sombra al texto.
- **color:** define el color del texto.

```
● ● ●

p {
    text-align: center;
    text-decoration: underline;
    text-indent: 2em;
    text-transform: uppercase;
    letter-spacing: 1px;
    word-spacing: 2px;
    text-shadow: 2px 2px 2px #aaa;
    color: #333;
}
```



CSS

Módulo XII –

Position

Página oficial en MDN

[https://developer.mozilla.org/es/docs/
Web/CSS/position](https://developer.mozilla.org/es/docs/Web/CSS/position)

La propiedad **position** en CSS se utiliza para controlar el método de posicionamiento de un elemento dentro de su contenedor. Esta propiedad define cómo se colocan y se solapan los elementos, permitiendo un control más preciso sobre la disposición de los elementos en la página.

- **static:** este es el valor por defecto. Los elementos se posicionan de acuerdo con el flujo normal del documento.
- **relative:** el elemento se posiciona en relación con su posición original en el flujo del documento.
- **absolute:** el elemento se posiciona en relación con su contenedor padre más cercano que tiene un posicionamiento distinto a **static**.
- **fixed:** el elemento se posiciona en relación con la ventana del navegador y permanece fijo al hacer scroll.
- **sticky:** es una mezcla entre **relative** y **fixed**. El elemento se trata como **rela-**

tive hasta que su bloque contenedor alcanza un punto de desplazamiento determinado en la pantalla.

Flow

Cuando hablamos del flujo (*Flow*) podemos definir que cada **position** tiene uno concreto:

- **Position static:** flujo normal o *normal flow*
- **Position relative:** flujo impuesto o *removed flow*
- **Position absolute:** flujo impuesto o *removed flow*
- **Position fixed:** flujo impuesto o *removed flow*



```
.elemento-static {  
    position: static;  
}  
  
.elemento-relative {  
    position: relative;  
    top: 10px;  
    left: 20px;  
}  
  
.elemento-absolute {  
    position: absolute;  
    top: 0;  
    right: 0;  
}  
  
.elemento-fixed {  
    position: fixed;  
    bottom: 0;  
    left: 0;  
}  
  
.elemento-sticky {  
    position: sticky;  
    top: 0;  
}
```



CSS

Módulo XIII–

Flex/Flexbox

Página oficial en MDN

https://developer.mozilla.org/es/docs/Learn/CSS/CSS_layout/Flexbox

Flexbox, o el modelo de caja flexible, es un enfoque moderno en CSS para el diseño de estructuras complejas y alineación de elementos en una forma más predecible y eficiente en comparación con los modelos de diseño tradicionales. Se orienta principalmente a proporcionar una distribución óptima del espacio a lo largo de un eje principal, que puede ser tanto horizontal como vertical.

El uso de **Flexbox** facilita la creación de diseños responsivos sin tener que recurrir a técnicas como flotados o posicionamiento, lo cual puede resultar complicado y propenso a errores. Es especialmente útil en aplicaciones web complejas y en pequeños componentes.

Propiedades *container*

Para utilizar **Flexbox**, se define un contenedor **flex** utilizando la propiedad **dis-**

play con el valor **flex** o **inline-flex**. A continuación se describen las propiedades aplicables a los contenedores **flex**:

- **display**: define un elemento como bloque flexible (**flex**) o bloque flexible en línea (**inline-flex**).
- **flex-direction**: define la dirección principal en la que se quieren alinear los elementos flexibles.
- **flex-wrap**: define si los elementos flexibles deben saltar de línea o no.
- **flex-flow**: es una propiedad abreviada para **flex-direction** y **flex-wrap**.
- **justify-content**: alinea los elementos hijos a lo largo del eje principal.
- **align-items**: alinea los elementos hijos a lo largo del eje transversal.
- **align-content**: alinea las líneas flex cuando el espacio del contenedor excede al de los elementos.



```
.contenedor-flex {  
    display: flex;  
    flex-direction: row;  
    flex-wrap: nowrap;  
    justify-content: space-between;  
    align-items: center;  
    align-content: stretch;  
}
```

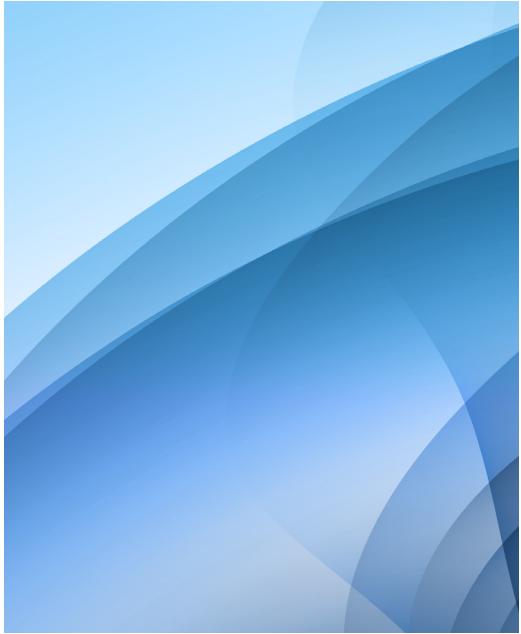
Propiedades Item

Las propiedades de los elementos flexibles permiten ajustar el comportamiento de estos dentro de un contenedor **flex**:

- **order**: define el orden de aparición del elemento en el contenedor.
- **flex-grow**: define la habilidad del elemento para crecer si es necesario.
- **flex-shrink**: define cómo debe reducirse el tamaño del elemento.
- **flex-basis**: define el tamaño base del elemento antes de distribuir el espacio restante.
- **flex**: es una propiedad abreviada para **flex-grow**, **flex-shrink** y **flex-basis**.
- **align-self**: permite la alineación predeterminada del elemento o anula la **align-items** del contenedor.



```
.item-flex {  
    order: 2;  
    flex-grow: 1;  
    flex-shrink: 2;  
    flex-basis: auto;  
    flex: 1 2 auto;  
    align-self: center;  
}
```



CSS

Módulo XIV–

Grid

Página oficial en MDN

[https://developer.mozilla.org/es/docs/
Web/CSS/grid](https://developer.mozilla.org/es/docs/Web/CSS/grid)

CSS Grid Layout es un sistema bidimensional de diseño que transforma el contenedor en una «cuadrícula» imaginaria donde se pueden posicionar elementos de manera más flexible y controlada. **Grid** permite crear *layouts* complejos y responsivos con una sintaxis más clara y menos propensa a errores, lo que lo hace una herramienta poderosa y eficiente para diseñadores y desarrolladores.

A diferencia de otros sistemas de diseño, como **Flexbox**, que se centra en la distribución a lo largo de un eje, **Grid** permite controlar simultáneamente las filas y columnas de la estructura, facilitando la creación de diseños que antes requerían el uso de múltiples herramientas y técnicas complicadas.

Propiedades *container*

Para convertir un elemento en un contenedor **Grid**, se usa la propiedad **display** con el valor **grid** o **inline-grid**. Aquí se describen las propiedades aplicables a los contenedores **Grid**:

- **display**: establece el elemento como una cuadrícula (**grid**) o cuadrícula en línea (**inline-grid**).
- **grid-template-columns / grid-template-rows**: define el tamaño de las columnas y filas.
- **grid-template-areas**: define áreas y las asocia a un nombre.
- **grid-template**: es una propiedad abreviada para **grid-template-rows**, **grid-template-columns** y **grid-template-areas**.
- **grid-column-gap / grid-row-gap**: define el espacio entre las columnas y filas.
- **grid-gap**: es una propiedad abreviada para **grid-column-gap** y **grid-row-gap**.
- **justify-items**: alinea los elementos a lo largo del eje de las columnas.
- **align-items**: alinea los elementos a lo largo del eje de las filas.
- **place-items**: es una propiedad abreviada para **align-items** y **justify-items**.
- **justify-content**: alinea la cuadrícula a lo largo del eje de las columnas.
- **align-content**: alinea la cuadrícula a lo largo del eje de las filas.
- **place-content**: es una propiedad abreviada para **align-content** y **justify-content**.



```
.contenedor-grid {  
    display: grid;  
    grid-template-columns: 1fr 2fr;  
    grid-template-rows: auto;  
    grid-gap: 10px;  
    justify-items: center;  
    align-items: start;  
}
```

Propiedades Item

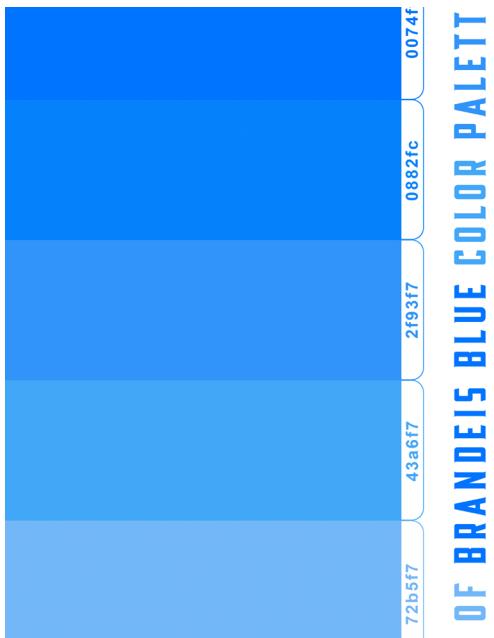
Las propiedades de los elementos **Grid** permiten controlar la ubicación y el tamaño de los items dentro del contenedor **Grid**:

- **grid-column-start / grid-column-end / grid-row-start / grid-row-end**: define la ubicación de un elemento en la cuadrícula.
- **grid-column / grid-row**: son propiedades abreviadas para **grid-column-start / grid-column-end** y **grid-row-start / grid-row-end**, respectivamente.

- **grid-area**: define el nombre del área al cual pertenece el elemento o especifica sus posiciones en la cuadrícula.
- **justify-self**: alinea un elemento dentro de su área en el eje de las columnas.
- **align-self**: alinea un elemento dentro de su área en el eje de las filas.
- **place-self**: es una propiedad abreviada para **align-self** y **justify-self**.



```
.item-grid {  
    grid-column: 1 / 3;  
    grid-row: 1 / 2;  
    grid-area: header;  
    justify-self: center;  
    align-self: start;  
}
```



CSS

Módulo XV–

Color

Página oficial en MDN

https://developer.mozilla.org/es/docs/Web/CSS/color_value

Color

El uso del color en CSS permite personalizar el aspecto visual de un sitio web, afectando tanto el texto como los elementos gráficos. Existen diferentes maneras de especificar colores en CSS, y estas pueden ser utilizadas según las necesidades del diseñador.

- **color:** define el color del texto de un elemento.
- **border-color:** define el color del borde de un elemento.

Especificación de colores:

- **Palabras clave:** como red, blue, green, etc.
- **Hexadecimal:** como #FF5733, #333, etc.
- **RGB:** como rgb(255, 87, 51).
- **RGBA:** como rgba(255, 87, 51, 0.5) (incluye canal alfa para opacidad).
- **HSL:** como hsl(12, 100%, 60%).
- **HSLA:** como hsla(12, 100%, 60%, 0.5)



```
.parrafo {  
    color: #FF5733; /* Color del texto */  
    border-color: rgba(255, 87, 51, 0.5); /* Color del borde con opacidad */  
}
```

Background-color

El color de fondo de un elemento se puede especificar utilizando la propiedad **background-color**. Ésta permite aplicar un color sólido que cubre el área del contenido, el **padding** y el borde del elemento.

- **background-color**: Define el color de fondo de un elemento.

Especificación de colores: similar a la propiedad **color**, puede utilizar palabras clave, valores hexadecimales, RGB, RGBA, HSL, y HSLA.



```
.caja {  
    background-color: hsl(200, 100%, 50%); /* Color de fondo */  
}
```

Opacidad

La propiedad **opacity** en CSS permite ajustar la opacidad de un elemento, lo cual determina cuán «transparente» aparece dicho elemento en la página.

- **opacity**: establece el nivel de opacidad de un elemento, con valores que van desde **0** (completamente transparente) hasta **1** (completamente opaco).



```
.imagen-transparente {  
    opacity: 0.5; /* Semi-transparente */  
}
```



CSS

Módulo XVI–

Imágenes de fondo

Página oficial de MDN

<https://developer.mozilla.org/en-US/docs/Web/CSS/background>

La propiedad **background-** en CSS permite establecer una o más imágenes como fondo de un elemento. Al usar imágenes de fondo, se pueden crear efectos visuales interesantes y mejorar la experiencia del usuario sin la necesidad de recurrir a imágenes incrustadas en el HTML.

Las propiedades que están asociadas a ella son:

- **background-image:** define una o más imágenes a utilizar como fondo.
- **background-repeat:** establece si/how una imagen de fondo se repetirá.
- **background-size:** define el tamaño de la imagen de fondo.
- **background-position:** establece la posición inicial de una imagen de fondo.
- **background-attachment:** determina si una imagen de fondo se desplazará con el resto de la página o quedará fija.

- **background-clip**: define hasta dónde llegará el fondo (borde, padding, contenido).
- **background-origin**: establece la caja de referencia para la posición y el tamaño del fondo.



```
.div-con-fondo {  
    background-image: url('imagen.jpg'); /* Imagen de fondo */  
    background-repeat: no-repeat; /* La imagen no se repite */  
    background-size: cover; /* La imagen cubre completamente el div */  
    background-position: center; /* La imagen se centra en el div */  
    background-attachment: fixed; /* La imagen de fondo se fija en la vista */  
    background-clip: border-box; /* El fondo se extiende */  
}
```



CSS

Módulo XVII–

Efectos visuales

Degrados CSS

Página oficial de MDN

<https://developer.mozilla.org/en-US/docs/Web/CSS/gradient>

Los degradados en CSS proporcionan la capacidad de mostrar transiciones suaves entre dos o más colores. Esto permite crear efectos visuales ricos sin necesidad de imágenes y existen varios tipos:

- **linear-gradient:** crea un degradado lineal.
- **radial-gradient:** crea un degradado radial.
- **conic-gradient:** crea un degradado cónico.
- **repeating-linear-gradient:** crea un degradado lineal repetitivo.
- **repeating-radial-gradient:** crea un degradado radial repetitivo.



```
.div-degradado {  
    background: linear-gradient(to right, red, orange);  
}
```

Sombras

Página oficial de MDN

<https://developer.mozilla.org/en-US/docs/Web/CSS/box-shadow>

CSS permite aplicar sombras a elementos, creando una sensación de profundidad y dimensión.

- **box-shadow:** aplica una sombra a la caja de un elemento.
- **text-shadow:** aplica una sombra al texto de un elemento.



```
.div-sombra {  
    box-shadow: 5px 5px 10px rgba(0, 0, 0, 0.5);  
    text-shadow: 2px 2px 4px rgba(0, 0, 0, 0.7);  
}
```

Modos de fusión

Página oficial de MDN

<https://developer.mozilla.org/en-US/docs/Web/CSS/mix-blend-mode>

Los modos de fusión determinan cómo se combinan los píxeles de dos elementos que se superponen.

- **mix-blend-mode:** define cómo se mezcla el contenido de un elemento con su fondo.



```
.div-fusion {  
    mix-blend-mode: multiply;  
}
```

Filtros

Página oficial de MDN

<https://developer.mozilla.org/es/docs/Web/CSS/filter>

CSS proporciona filtros que permiten ajustar la apariencia de los elementos de manera no destructiva.

- **filter:** aplica operaciones gráficas (como desenfoque o brillo) a un elemento.



```
.div-filtro {  
    filter: blur(4px) brightness(150%);  
}
```

Transformaciones

Página oficial de MDN

<https://developer.mozilla.org/en-US/docs/Web/CSS/transform>

Las transformaciones en CSS permiten modificar la posición, tamaño y forma de un elemento.

- **transform:** aplica una transformación 2D o 3D a un elemento.
- **transform-origin:** cambia el punto de origen de la transformación.



```
.div-transformado {  
    transform: rotate(45deg) scale(1.5);  
    transform-origin: center;  
}
```



CSS

Módulo XVIII–

Animaciones

Página oficial de MDN

<https://developer.mozilla.org/en-US/docs/Web/CSS/animation>

Las animaciones en CSS permiten crear transiciones fluidas y dinámicas entre diferentes estilos de un elemento.

Se pueden utilizar para mejorar la interactividad y la experiencia del usuario en una página web y a diferencia de las simples transiciones CSS, las animaciones ofrecen un control más preciso sobre los diferentes estados de un elemento durante el período de la animación.

Animation

Las animaciones en CSS se crean combinando dos componentes principales: la propiedad **animation** y la regla **@keyframes**.

- **animation-name:** define el nombre de la animación.
- **animation-duration:** establece cuánto tiempo tardará en completarse una iteración de la animación.

- **animation-timing-function:** define la velocidad de la animación.
- **animation-delay:** especifica el tiempo que debe esperar antes de iniciar la animación.
- **animation-iteration-count:** define cuántas veces se repetirá la animación.
- **animation-direction:** establece si la animación debe reproducirse en reversa, alternar, etc.
- **animation-fill-mode:** define el estado de la animación cuando no está reproduciéndose.
- **animation-play-state:** define si la animación está en ejecución o pausada.



```
.div-animado {
    animation-name: deslizar;
    animation-duration: 2s;
    animation-timing-function: ease-in-out;
    animation-iteration-count: infinite;
    animation-direction: alternate;
}
```

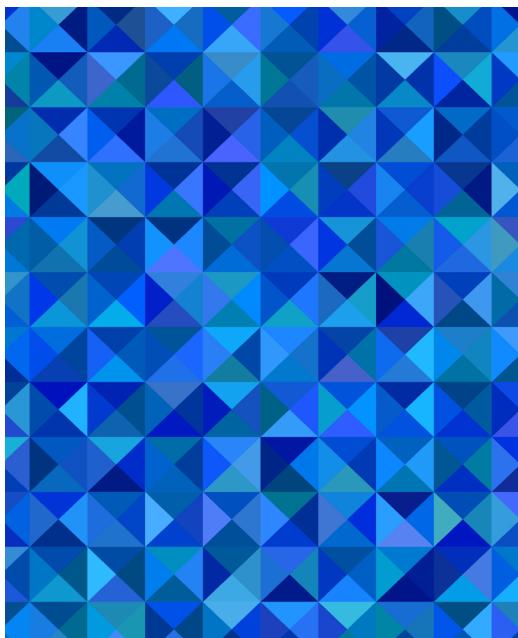
Keyframes

La regla **@keyframes** se utiliza para crear animaciones estableciendo diferentes puntos (keyframes) a lo largo de la animación.

- **from/to:** establece el punto de inicio (0%) y finalización (100%) de la animación.
- **% (porcentaje):** especifica los estilos en diferentes puntos a lo largo de la animación.



```
@keyframes deslizar {
    from {
        transform: translateX(0);
    }
    to {
        transform: translateX(100px);
    }
}
```



CSS

Módulo XIX-

Media Queries

Página oficial de MDN

https://developer.mozilla.org/es/docs/Web/CSS/CSS_media_queries/Using_media_queries

Los *Media Queries* son una característica de CSS3 que permite aplicar estilos condicionalmente en función de ciertas propiedades del dispositivo de visualización, como su ancho, alto, resolución, etc. Esto hace posible la creación de diseño responsive, donde el contenido de una página web se ajusta y optimiza para diferentes tamaños de pantalla y dispositivos.

El uso de *Media Queries* ha revolucionado la forma en que se diseñan las páginas web, permitiendo a los desarrolladores crear experiencias de usuario más fluidas y adaptativas. Con esta técnica, se puede cambiar el estilo de la página de acuerdo a las características del dispositivo que se esté utilizando para visualizarla, garantizando así una experiencia de usuario coherente en diferentes plataformas.

Breakpoints

Los *breakpoints* son puntos específicos de medida, como un ancho de pantalla, donde el contenido de una página cambia su disposición para adaptarse mejor al espacio disponible y debes tomarlos como una referencia, no como medidas absolutas.

- **Extra pequeño (xs):** $<360\text{px}$, para teléfonos móviles pequeños
- **Pequeño (sm):** $\geq 480\text{px}$, para teléfonos móviles
- **Mediano (md):** $\geq 768\text{px}$, para *tablets* en orientación horizontal y algunos ordenadores.
- **Grande (lg):** $\geq 1200\text{px}$, para ordenadores de escritorio
- **Extra grande (xl):** $\geq 1600\text{px}$, para pantallas grandes.

Media-Query

La regla **@media** en CSS se utiliza para definir *Media Queries*. Dentro de esta regla, se especifican los estilos que deben aplicarse condicionalmente.

Tipos de *Media Queries*:

- **max-width:** aplica estilos para dispositivos con una anchura de pantalla máxima especificada.
- **min-width:** aplica estilos para dispositivos con una anchura de pantalla mínima especificada.
- **orientation:** aplica estilos según la orientación del dispositivo (*landscape* o *portrait*).
- **resolution:** aplica estilos según la resolución de la pantalla.

```
① ② ③  
 @media (max-width: 576px) {  
     body {  
         background-color: lightblue;  
     }  
 }
```



JAVASCRIPT

Módulo XX-

Introducción

Página oficial en MDN

<https://developer.mozilla.org/es/docs/Web/JavaScript>

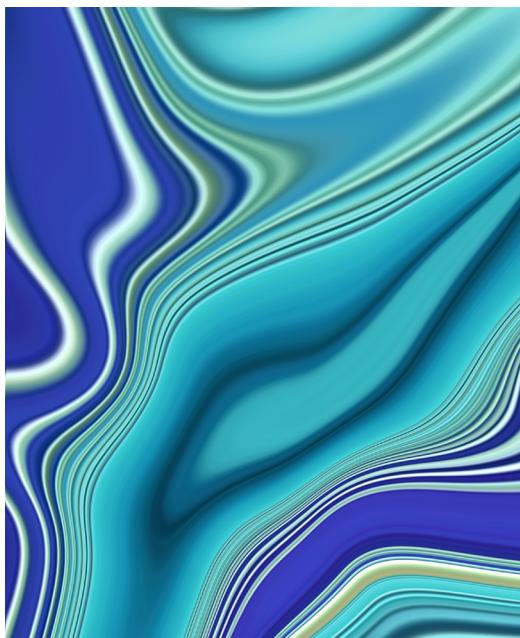
JavaScript es un lenguaje de programación de alto nivel, interpretado y dinámicamente tipado, que ha evolucionado desde su creación para manipular páginas web hasta convertirse en una herramienta omnipresente en el desarrollo moderno, tanto en el lado del cliente como del servidor.

Originalmente diseñado por Brendan Eich en 1995, JavaScript se ha estandarizado bajo el nombre de ECMAScript. ECMAScript actúa como la base sobre la cual se construye JavaScript, y sus especificaciones, publicadas por ECMA International, han ido introduciendo a lo largo de los años mejoras y nuevas características que han enriquecido y modernizado el lenguaje.

Modos de uso: *Floppy Mode* y «*use strict*»

JavaScript puede operar en dos modos distintos: el modo tradicional, conocido informalmente como «*Floppy Mode*», y el modo estricto, activado con la directiva «*use strict*».

El modo *Floppy* es más permisivo y tolerante con ciertos errores, autocorrigiéndolos en tiempo de ejecución. Por otro lado, el modo estricto introduce restricciones adicionales y convierte algunos errores silenciosos en errores explícitos, facilitando la detección de problemas en las etapas iniciales del desarrollo. Este último modo, al ser más riguroso, ayuda a los desarrolladores a escribir código más limpio y predecible, al tiempo que previene ciertas acciones potencialmente problemáticas.



JAVASCRIPT

Módulo XXI–

Constantes, variables y *scope*

Variables

Página oficial en MDN

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let>

Los usamos para guardar (almacenar) un valor que puede cambiar a lo largo del código y su declaración estará relacionada con donde se pueden usar, es decir su *scope*.

```
● ● ●

let nombre          // Declaración de la variable
nombre = 'Timmy'   // Asignación de la variable

// Declaración múltiple
let nombre = 'Timmy' , apellidos = 'Tintor'
```

Constantes

Página oficial en MDN

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const>

Las usamos para guardar (almacenar) un valor que no cambia y su declaración estará relacionada con donde se pueden usar, es decir su *scope*.

```
const nombre = 'Timmy' // Asignación de la variable  
// Declaración múltiple  
const nombre = 'Timmy' , apellidos = 'Tintor'
```

En el caso tanto de las variables **let** como de las constantes recuerda que no podemos acceder a ellas antes de declararlas; es decir, no tienen *hoisting* sino TDZ (*Temporal Dead Zone*).

Scope

Página oficial en MDN

<https://developer.mozilla.org/es/docs/Glossary/Scope>

Define desde dónde se puede acceder, y manipular, tanto a variables como a constantes dentro del código de JavaScript, en otras palabras: desde dónde podemos acceder a ellas para usarlas. Existen varios tipos:

- Global
- Local
- Bloque

```
let variable1 // Declaración global  
  
function accion(){  
  
    let variable2 // Declaración Local  
  
    for( let i ; ... ; ... ){ // Declaración de bloque  
  
    }  
  
}
```

SCOPE GLOBAL

Las variables (**let, var**) o constantes declaradas fuera de una función son de tipo GLOBAL. Las variables globales son accesibles en todas las funciones y cualquier parte del *script* de esa misma página.

SCOPE LOCAL

Las variables (**let, var**) o constantes declaradas dentro de una función son de tipo LOCAL. Las variables locales solo pueden utilizarse dentro de la función. Ya que las

variables locales solo son reconocidas por su función, se permite reutilizar el mismo nombre en diferentes funciones.

Además, estas variables se crean cuando la función se ejecuta y se eliminan una vez la función se ha completado.

SCOPE BLOQUE

Las variables (**let**) o constantes declaradas dentro de bloques como condicionales o bucles **for** son de tipo BLOQUE y sólo funcionarán dentro de él.



JAVASCRIPT

Módulo XXII–

Tipos de datos

Página oficial en MDN

https://developer.mozilla.org/es/docs/Web/JavaScript/Data_structures

Es la información que podemos usar dentro de JavaScript y tienen características propias.

Veremos los tipos de datos:

- Primitivos
- Referencia

```
● ● ●

let edad      = 20                      // Tipo Number
let nombre    = 'Pepe'                   // Tipo String
let usuario  = {nombre: 'Ana', edad: 30}  // Tipo Object
let latitud   = 5.1224213134123        // Tipo Number (Float)
let peso      = 1.853                    // Tipo Number (Double)
let activo   = true                     // Tipo Boolean
let nada     = null                     // Tipo Null
let datos    = ['Manzana', 'Melón', 'Naranja'] // Tipo Array
```

Primitivos

STRING

Un *string* (o cadena de texto) es una serie de caracteres, como, por ejemplo, "Hola, Mundo". Los *string* se escriben siempre entre comillas (dobles o simples).



```
// Dobles en el String, simples en interior
let cadena1 = 'Estas 'comillas' funcionan'
// Simples en el String, Dobles en interior
let cadena2 = "Estas 'comillas' funcionan"

let cadena3 = 'Así 'no' se usan' // ERROR
let cadena4 = "Así "no" se usan" // ERROR
```

También pueden escribirse unas comillas dentro de otras, siempre y cuando no coincidan entre sí dentro de la cadena:



```
let nombre1 = "Pepe Pepítez" // Comillas Dobles
let nombre2 = 'Pepe Pepítez' // Comillas Simples
```

NUMBER

El tipo de dato **Number** son números y representan cualquier tipo de número (enteros, con decimales, negativos...)



```
let edad1      = 32           // Asignamos un número a una variable
let edad2      = Number(32)    // Asignamos un número con Number()
let precio     = 15.4         // Con decimales
let negativo   = -15          // Negativos
```

BOOLEAN

Es un tipo de dato con sólo dos posibles valores, *true* o *false*, y tendrán mucha importancia al ver los condicionales.



```
true    // Verdadero  
false   // Falso
```

Referencia

Se guardan en memoria, son mutables y existen varios tipos:

- Arrays
- Objetos

ARRAYS

Es un tipo de dato que puede guardar múltiples valores; es decir, una lista de valores.

Siempre debe especificarse con el símbolo **[] (corchetes)** y cada elemento debe estar separado por **,** (**comas**).

Más adelante entraremos en detalle sobre los *arrays*.



```
['lunes','martes']           // Array de strings  
[1,34,54,21]                // Array de numbers  
[true, false,true, true]     // Array de booleans  
[{ nombre : 'Timmy'},{ nombre : 'Edu'}] // Array de Objetos
```

OBJETOS

Un *object* (u objeto) siempre debe especificarse con el símbolo **{ } (llaves)** y cada propiedad del elemento debe estar separado por **,** (**comas**) y pares de “**nombre: valor**” (en inglés “**key : value**”).

Más adelante entraremos en detalle sobre los *objetos*.



```
let alumno = {  
    propiedad : '', // Es como una variable  
    metodo     : ()=>{} // Es como una función  
}  
  
alumno.propiedad // Así accedemos a una propiedad  
alumno.metodo() // Así accedemos a una método
```

Otros tipos

Existen otros tipos de datos que nos encontraremos cuando programemos en JavaScript:

- **Undefined:** un *undefined* (o valor indefinido) ocurre cuando una variable no posee ningún valor.
- **Empty:** un *empty* (o vacío) no tiene nada que ver con un “*undefined*” ya que un *empty* ocurre cuando el tipo de dato es *string* pero no posee valor alguno.
- **Null:** un *null* (o nulo) es literalmente “nada”. Se supone que es algún valor que ni siquiera existe.
- **NaN:** *Not a Number*.



```
let variable = undefined // (PRIMITIVO) Undefined: Valor no definido  
let variable = NaN       // NaN: Operación matemática no válida  
let variable = null      // Null : Valor vacío  
let variable = ''         // Empty
```

Conversión de tipos

En ciertos casos nos veremos en la necesidad de convertir un tipo de dato en otro. Por ejemplo, si queremos hacer algún cálculo, no podremos sumar *strings*, por ejemplo:



```
'1' + '2' = '12' // Unirá las dos cadenas ya que están entrecomilladas
1    + 2    = 3    // Hará el cálculo porque detecta que son números

/* Ejemplo de conversión */
let numero = 12
let texto  = "12"

numero.toString() // Para convertir a string
parseInt(texto) // Para convertir a Number(integer)
parseFloat(texto) // Para convertir a Number(float)
```

Truthy/ Falsy

Página oficial en MDN

<https://developer.mozilla.org/en-US/docs/Glossary/Falsy>

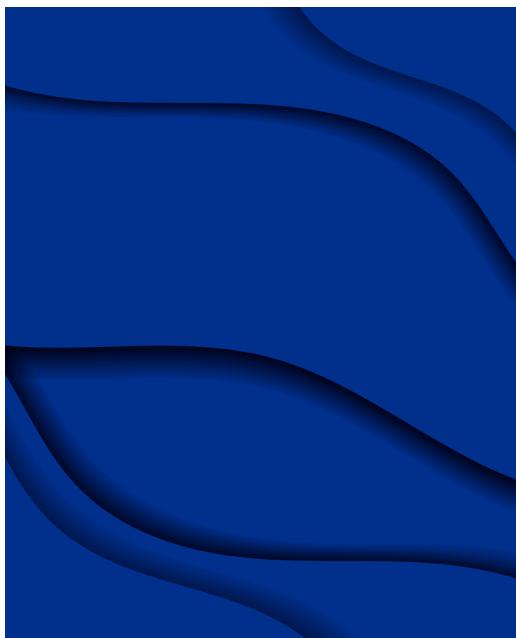
<https://developer.mozilla.org/es/docs/Glossary/Truthy>

Algunos de los valores son evaluados automáticamente como verdaderos o falsos en contextos booleanos, como las declaraciones condicionales

Los valores que son evaluados como falsos en un contexto booleano se conocen como **Falsy**. Estos incluyen **false**, **0**, **""** (cadena vacía), **null**, **undefined** y **NaN**.

Por otro lado, los valores que son evaluados como verdaderos en un contexto booleano se denominan **Truthy**. Cualquier valor que no sea *Falsy* es considerado *Truthy*.

Esto incluye objetos, arrays, funciones y cualquier número o cadena no vacía. Entender la diferencia entre valores *Truthy* y *Falsy* es crucial para la escritura efectiva de condicionales y la evaluación lógica en JavaScript.



JAVASCRIPT

Módulo XXIII–

Arrays

Página oficial en MDN

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array

Los arrays se utilizan para almacenar múltiples valores en una misma variable.

Son un tipo espacial de “contenedor”, ya que puede almacenar diferentes valores al mismo tiempo. Si tenemos una lista de elementos, podemos almacenarlos todos sin necesidad de crear variables independientes.

```
● ● ●

['lunes', 'martes'] // Array de strings
[1, 34, 54, 21] // Array de numbers
[true, false, true, true] // Array de booleans
[ { nombre : 'Timmy' }, { nombre : 'Edu' } ] // Array de Objetos
```

Creación

La forma más fácil de utilizar un array es creando un array literal o la posibilidad de utilizar la palabra reservada “new”.



```
let nombreArray = [elemento1, elemento2, ... ]
```

Acceder a los elementos de un array

Podemos acceder a datos concretos de un *array* haciendo referencia a su posición entre los índices del mismo.



```
let cubiertos = ["tenedor", "cuchara", "cuchillo"]
console.log(cubiertos[0]) // Muestra "tenedor"
console.log(cubiertos[1]) // Muestra "cuchara"
console.log(cubiertos[2]) // Muestra "cuchillo"
console.log(cubiertos)    // Muestra ["tenedor", "cuchara", "cuchillo"]
```

Propiedades

Podemos acceder al número de valores de un *array* gracias al atributo *length*.



```
let semana = ['lunes', 'martes', 'miércoles']
semana.length // El resultado sería "3"

let animales = ['🐶', '🐱']
animales.length // El resultado sería "2"
```

Métodos de arrays

Los *arrays* tienen múltiples métodos que podemos usar, revisemos algunos de los más usados.

UNSHIFT

Nos permite añadir un valor al *inicio* del *array*.



```
let emojis = ['😊', '💩']  
emojis.unshift('😡') // El resultado es [ '😡', '😊', '💩' ]
```

PUSH

Nos permite añadir un valor al *final* del *array*.



```
let emojis = ['😊', '💩']  
emojis.push('😡') // El resultado es [ '😊', '💩', '😡' ]
```

SPLICE

El método **splice** puede usarse para añadir nuevos elementos a un *array* o también puede usarse **splice()** para eliminar elementos del *array* sin dejar “huecos” en el *array*.



```
miArray.splice(2, 0, "ElementoN1")  
// El primer parámetro es la posición donde se insertarán los elementos  
// El segundo parámetro es el número de elementos que se eliminarán  
// El tercer parámetro es el elemento que se insertará en la posición indicada  
  
miArray.splice(0, 1) // Elimina desde la posición 0, 1 elemento  
miArray.splice(0, 2) // Elimina desde la posición 0, 2 elementos
```

INDEXOF

El método **indexOf()** busca un valor concreto dentro de un conjunto de valores (*array*, *texto...*) y devuelve su posición.

Si el valor de un **indexOf()** no se encuentra devolverá el valor **-1**, en caso positivo devolverá el índice de su posición.



```
alimentos.indexOf("Lechuga") // Muestra la posición de "Lechuga" es decir: 0  
alimentos.indexOf("Tomate") // Muestra la posición de "Tomate" es decir: 1
```

REVERSE

Nos permite darle la vuelta a los valores del *array*.



```
let emojis = ['😊', '💩', '😡']  
emojis.reverse() // El resultado es [ '😡', '💩', '😊' ]
```

JOIN

Nos permite generar un *string* uniendo los valores de un *array*.



```
let emojis = ['😊', '💩', '😡', '🍏', '🦄']  
emojis.join() // Devuelve '😊💩😡🍏🦄'  
emojis.join('') // Devuelve '😊💩😡🍏🦄'  
emojis.join('-') // Devuelve '😊-💩-😡-🍏-🦄'
```

FOREACH

Nos permite recorrer un *array* de elementos dándonos acceso tanto al valor como a la posición de ese valor. Este método no devuelve nada.



```
let emojis = ['😊', '💩', '😡', '🍏', '🦄']  
  
emojis.forEach( ( cadaEmoji, i ) => {  
    console.log( cadaEmoji ) // Este parámetro sería el valor  
    console.log( i ) // Este parámetro sería la posición  
})
```

MAP

Nos permite recorrer un *array* de elementos dándonos acceso tanto al valor como a la posición de ese valor. Este método devuelve todos los valores (si se lo aplicamos).



```
let emojis = ['😊', '💩', '😡', '🍏', '🦄']
emojis.map( cadaEmoji => cadaEmoji + '💥' )
// El resultado es [ '😊💥', '💩💥', '😡💥', '🍏💥', '🦄💥' ]
```

FIND

Nos permite recorrer un *array* de elementos dándonos acceso al valor.

Cuando encuentra un valor con la condición que le damos, devuelve ese valor y para de buscar.



```
let emojis = [ '😊', '💩', '😡💥', '🍏', '🦄' ]
emojis.find( cadaEmoji => cadaEmoji.includes('💥') )
// El resultado es '😡💥'
```

SOME

Nos permite recorrer un *array* de elementos tenemos acceso al valor.

Cuando encuentra un valor con la condición que le damos, devuelve *true/false* y para de buscar.



```
let emojis = [ '😊', '💩', '😡💥', '🍏', '🦄' ]
emojis.some( cadaEmoji => cadaEmoji.includes('💥') )
// El resultado es "true"
```

FILTER

Nos permite recorrer un *array* de elementos dándonos acceso al valor.

Cuando encuentra un valor con la condición que le damos, devuelve el valor y sigue buscando.

```
● ● ●

let hogwarts = [
  { nombre : 'Harry'      , apellido : 'Potter'    , casa : 'Griffindor'},
  { nombre : 'Ron'        , apellido : 'Weasly'     , casa : 'Griffindor'},
  { nombre : 'Hermione'   , apellido : 'Granger'   , casa : 'Griffindor'},
  { nombre : 'Draco'      , apellido : 'Malfoy'    , casa : 'Slytherin'}
]

// Estamos buscando los alumnos que sean de la casa 'Griffindor'
let griffindor = hogwarts.filter( alumno => alumno.casa === 'Griffindor' )
```

DECONSTRUCCIÓN

Podemos sacar cada uno de los valores sin mencionar las posiciones de un array. Es decir podemos usar *nombres* para cada valor. Esto es importante para ReactJS.

```
● ● ●

const alumno = ['Timmy','García','Diseño web']

// Asigna a cada valor una variable
const [ nombre , apellido , curso ] = alumno

// Si dejamos un espacio vacío, ese valor no lo asigna
const [ nombre , , curso ] = alumno
```



JAVASCRIPT

Módulo XXIV–

Objetos

Página oficial en MDN

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Object

Un objeto es un conjunto de propiedades (variables) y métodos (funciones).

```
● ● ●

let alumno = {
    propiedad : '',      // Es como una variable
    metodo     : ()=>{} // Es como una función
}

alumno.propiedad // Así accedemos a una propiedad
alumno.metodo() // Así accedemos a una método
```

Propiedades

Los valores de las propiedades de un objeto pueden ser cualquier tipo de dato: **Strings**, **Boolean**, **Number**...



```
let nombreObjeto = {  
    propiedad : 'valor' ,  
    propiedad : 31 ,  
    propiedad : true ,  
    propiedad : ['valor','valor'],  
    propiedad : {  
        propiedad : '' ,  
        propiedad : '' ,  
    }  
}
```

ACCEDER A LAS PROPIEDADES



```
let alumno = {  
    nombre : 'Alex',  
    apellido : 'Tintor',  
    ciudad : { cp : '28004' , calle : 'Bermejo 32'}  
}  
  
alumno.nombre // Así accedemos al nombre  
alumno.apellido // Así accedemos al apellido  
alumno.ciudad.cp // Así accedemos al código postal  
alumno.ciudad.calle // Así accedemos a la calle
```

DECONSTRUCCIÓN DE PROPIEDADES

Nos permite guardar en una variable una propiedad de un objeto. Se suele usar mucho en React.



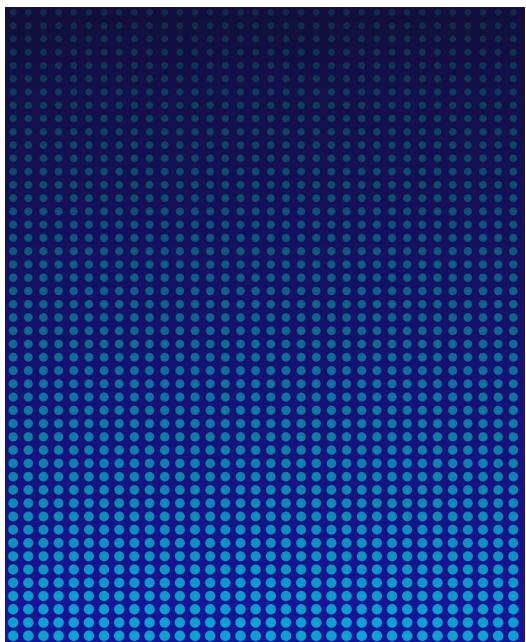
```
const alumno = {nombre : 'Timmy', apellido : 'García', curso : 'Web' }  
  
// Deconstruirmos el objeto en 3 constantes  
// ¡Importante! Las variables deben de llamarse igual que las propiedades  
const { nombre , apellido , curso } = alumno;  
  
// Cambiamos la prop nombre a propio1  
// Cambiamos la prop apellido a propio2  
const { nombre : propio1 , apellido : propio2 } = alumno;
```

METODOS

Los métodos son las funciones definidas dentro de un objeto.



```
let objeto1 = {  
    metodo1 : function(){}, // Con Función clásica  
}  
let objeto2 = {  
    metodo1 : () => {}, // Con Función Arrow  
}
```



JAVASCRIPT

Módulo XXV–

Funciones

Página oficial en MDN

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function

Una función es un bloque de código diseñado para realizar una tarea concreta. Solo se ejecutan cuando se las “invoca” (se las llama).

Cuando se desarrolla una aplicación compleja, es muy habitual utilizar una y otra vez las mismas instrucciones. Cuando una serie de instrucciones se repiten una y otra vez, se complica demasiado el código fuente de la aplicación, ya que:

- El código de la aplicación es mucho más largo porque muchas instrucciones están repetidas.
- Si se quiere modificar alguna de las instrucciones repetidas, se deben hacer tantas modificaciones como veces se haya escrito esa instrucción, lo que se convierte en un trabajo muy pesado y muy propenso a cometer errores.



```
function saludo(){
    console.log(`;Suscríbete!`)
}
saludo()
saludo()
saludo()
saludo()
```

Sintaxis

Las funciones se definen con la palabra reservada “*function*”, seguida de un nombre y, a su vez, seguido de un () (**paréntesis**). Los nombres de las funciones pueden contener letras, dígitos, barra baja y símbolo de dólar (al igual que las variables).

El paréntesis puede incluir parámetros separados por comas: (parametro1, parametro2...).

El código que debe ejecutarse una vez se invoque a la función se programará dentro del símbolo {} (**llaves**).

Invocación

El código dentro de la función se ejecutará solo si “algo” lo invoca (llama):

- Cuando las funciones de JavaScript alcanzan el punto de un *return*, la función dejará de ejecutarse. Si la función fue invocada, devolverá los datos que le pidamos.
- Cuando ocurre un evento (p. ej.: el usuario hace “click” sobre algo). Cuando se le invoca directamente desde el código JavaScript.
- Automáticamente (*self-invoked*).

Return

Cuando las funciones de JavaScript alcanzan el punto de un *return*, la función dejará de ejecutarse. Si la función fue invocada, devolverá los datos que le pidamos.

Ejemplo de invocación + *return*

```
● ● ●

// La función devuelve la suma de a y b
function sumar(a, b) {return a + b}

// Invoca a la función "sumar" con parámetros 9 y 6
let resultado = sumar(9, 6)
```

Tipos de funciones

Las funciones pueden ser definidas de diversas maneras como:

- Las **funciones nombradas** son declaradas con un nombre.
- Las **funciones anónimas** carecen de nombre y suelen utilizarse como argumento de otras funciones o como expresiones inmediatas.
- Las **funciones generadoras** se definen usando **function*** y permiten realizar operaciones de iteración pausada mediante el uso de **yield**.
- Las **funciones Arrow () => {}** introducidas en ES6 ofrecen una sintaxis concisa y no tienen su propio **this**, tomando el **this** del contexto circundante.
- Las **IIFE (Immediately Invoked Function Expression)** son funciones que se ejecutan inmediatamente después de ser definidas, encapsulando variables y protegiendo el alcance global

Cada tipo de función tiene su propio caso de uso, permitiendo a los desarrolladores escribir código más flexible y legible.



JAVASCRIPT

Módulo XXVI–

Operadores

Página oficial en MDN

<https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Operators>

Los operadores nos permiten realizar acciones con los diferentes tipos de datos y su funcionalidad depende tanto del tipo de dato como de la operación que queramos ejecutar.

Existen varios tipos de operadores en JavaScript:

- Asignación
- Aritméticos
- Lógicos
- Comparación

Asignación



```
variable = 'valor'      // Igualamos el valor
variable += 'valor'     // Sumamos el valor previo
variable -= 'valor'     // Restamos el valor previo
variable /= 'valor'     // Dividimos el valor previo
variable *= 'valor'     // Multiplicamos el valor previo
variable %= 'valor'     // Hacemos el módulo el valor previo
```

Aritméticos

```
● ● ●  
10 + 5 // Suma  
10 - 5 // Resta  
10 / 5 // División  
10 * 5 // Multiplicación  
10 % 5 // Módulo  
10++ // Incremento  
10-- // Decremento
```

Lógicos

Recuerda que algunos de estos operadores se usan con valores de JavaScript que consideramos tanto Truthy como Falsy.

```
● ● ●  
!true // Negación  
true && true // And  
true || true // Or
```

Verificación de tipos

```
● ● ●  
typeof variable // Devuelve el tipo de una variable  
instanceof variable // Devuelve “true” si un objeto es una instancia de otro
```



JAVASCRIPT

Módulo XXVII–

Fechas

Página oficial en MDN

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Date

JavaScript posee varios métodos integrados que nos permiten, de una manera muy sencilla, poder trabajar con fechas y horas. Esto resulta muy útil para poder controlar todo tipo de información (horas de conexión de usuarios, logs, cuenta-atrás...).

Estos métodos pueden utilizarse para obtener información sobre una fecha:

```
let fecha = new Date('July 20, 69 00:20:10')

fecha.getFullYear() // Año en formato de cuatro dígitos (aaaa)
fecha.getMonth()   // Mes como número (0-11)
fecha.getDate()    // Día como número (1-31)
fecha.getHours()   // Hora en formato 24 horas (0-23)
fecha.getMinutes() // Minutos (0-59)
fecha.getSeconds() // Segundos (0-59)
fecha.getMilliseconds() // Milisegundos (0-999)
fecha.getTime()    // La hora actual (en milisegundos)
fecha.getDay()     // El día de la semana como número (0-6)
Date.now()         // Fecha y hora actual
```



JAVASCRIPT

Módulo XXVIII–

Math

Página oficial en MDN

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Math

El objeto **Math** permite realizar tareas matemáticas de una manera muy ágil y sencilla.

Propiedades



```
Math.E      // El número de Euler
Math.PI     // El valor de Pi
Math.SQRT2  // La raíz cuadrada de 2
Math.LN2    // El logaritmo natural de 2
Math.LN10   // El logaritmo natural de 10
Math.LOG2E  // El logaritmo de la constante de Euler
Math.LOG10E // El logaritmo de la base 10 de la constante de Euler
```

Métodos

ROUND

Función para redondear un número al entero más cercano o a una cantidad específica de decimales.



```
Math.round(4.7) // Devuelve 5  
Math.round(4.4) // Devuelve 4
```

POW

Función para calcular la potencia de un número, elevando una base a un exponente dado.



```
Math.pow(8, 2) // Devuelve 64
```

RANDOM

Función para generar un número aleatorio en un rango determinado.



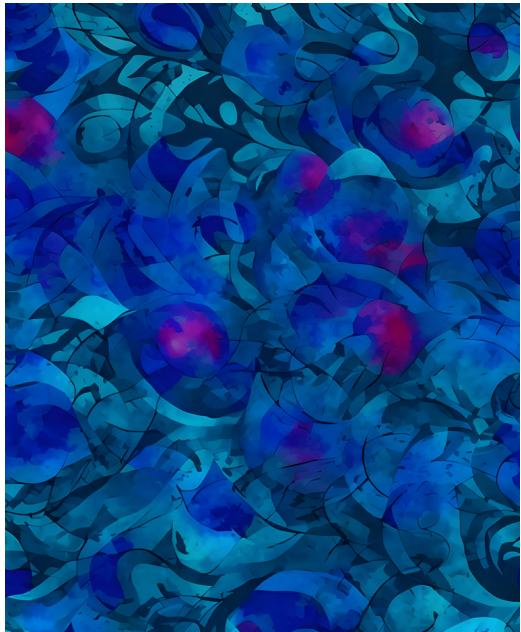
```
Math.random() // Devuelve un número aleatorio
```

SQRT

Función para calcular la raíz cuadrada de un número.



```
Math.sqrt(64) // Devuelve 8
```



JAVASCRIPT

Módulo XXIX–

Condiciones

Página oficial en MDN

<https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Statements/if...else>

Las sentencias condicionales se utilizan para realizar diferentes acciones y comprobaciones según se cumplan o no ciertos requisitos.

Cuando escribimos código, dependiendo de ciertas “decisiones”, desearemos que se ejecuten unas acciones u otras. Es ahí donde entran en juego las condiciones. Podemos crear condiciones para realizar las siguientes acciones dependiendo de lo que queramos comprobar:

- Usaremos “if” para especificar un bloque de código que se ejecutará si una condición específica se cumple.
- Usaremos “else” para especificar un bloque de código que se ejecutará si esa misma condición no se cumple.

- Usaremos “else if” para especificar una nueva condición para probar nuevamente si la primera condición no se cumple.
- Usaremos “switch” para especificar varios bloques alternativos de código que pueden ejecutarse (es una alternativa al “if” pero ofrece muchas más limitaciones y tarda más en ejecutarse, por lo que cada vez se usa menos en JavaScript).

If



```
if (condiciónSeCumple) {
    // Bloque de código que se ejecutará si se cumple la condición
}

// Ejemplo
if (hora < 14) {
    console.log(`Bienvenido a CEI`)
}
```

Else



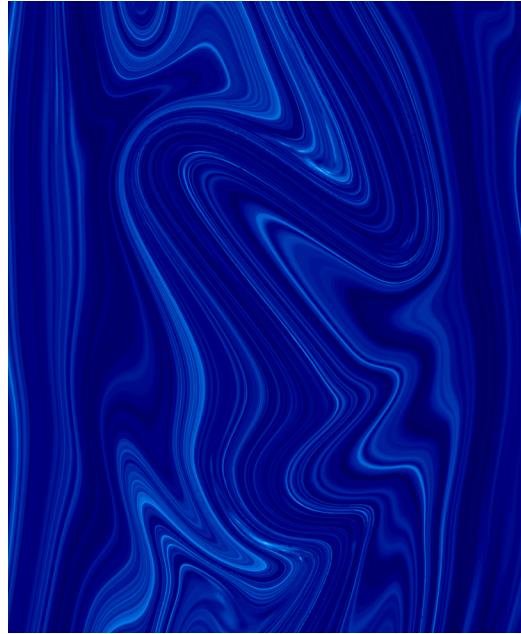
```
if (condiciónSeCumple) {
    // Bloque de código que se ejecutará si se cumple la condición
} else{
    // Bloque de código que se ejecutará si NO se cumple la condición
}
// Ejemplo:
if (hora < 14) {
    console.log(`Buenos días`)
} else{
    console.log(`Buenas tardes`)
}
```

Else if

```
● ● ●

if (condicion1) {
    // Bloque de código que se ejecutará si se cumple la condición
} else if( condicion2 ){
    // Bloque de código que se ejecutará si cumple la condicion2
} else{
    // Bloque de código que se ejecutará si NO cumple ninguna condición
}

// Ejemplo:
if (hora < 14){
    console.log(`Buenos días`)
} else if (hora < 20){
    console.log(`Buenas tardes`)
} else{
    console.log(`Buenas noches`)
}
```



JAVASCRIPT

Módulo XXX-

Bucles

Página oficial en MDN

https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Loops_and_iteration

Los bucles son prácticos, si deseamos ejecutar el mismo código una y otra vez, y cada vez con un valor diferente. Son bloques de código que se ejecutan un número X de veces. Existen diferentes tipos de bucles:

- **for**
- **while** (no muy usado en JavaScript pero sí en PHP)
- **do... while** (en desuso)
- **forEach()** (visto anteriormente en los arrays)

For

- Inicio del contador: se ejecuta (una vez) antes de la ejecución del bloque de código.
- Condición: define la condición que debe cumplirse para seguir ejecutando el código.

- Fin de Iteración: se ejecuta (cada vez) después de que el bloque de código se haya ejecutado.

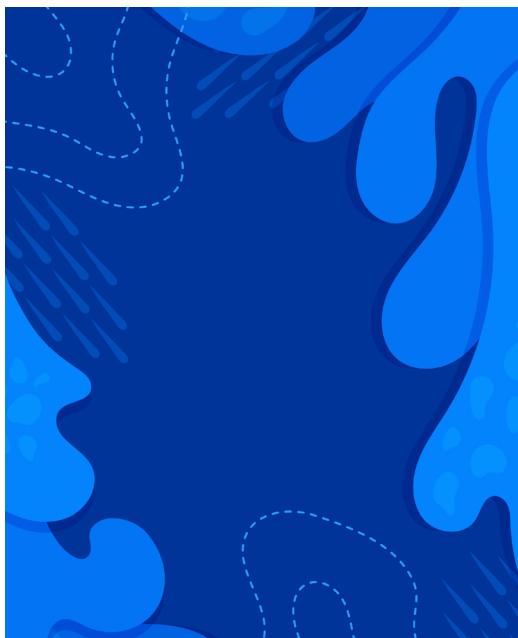


```
for (inicioDelContador; condicion; finDeLaIteraccion) {  
    // Bloque de código que se ejecutará  
}
```

EJEMPLO:



```
for (let i = 0; i < 10; i++) {  
    console.log(`El valor de i es: ${i}`)  
}
```



JAVASCRIPT

Módulo XXXI–

BOM

El BOM (*Browser Object Model*) permite a JavaScript comunicarse con el navegador. No existe un estándar oficial para el BOM, pero desde que los navegadores modernos han implementado (casi) los mismos métodos y propiedades para la interactividad con JavaScript se puede utilizar sin mayor inconveniente en todos los navegadores modernos..

Objeto window

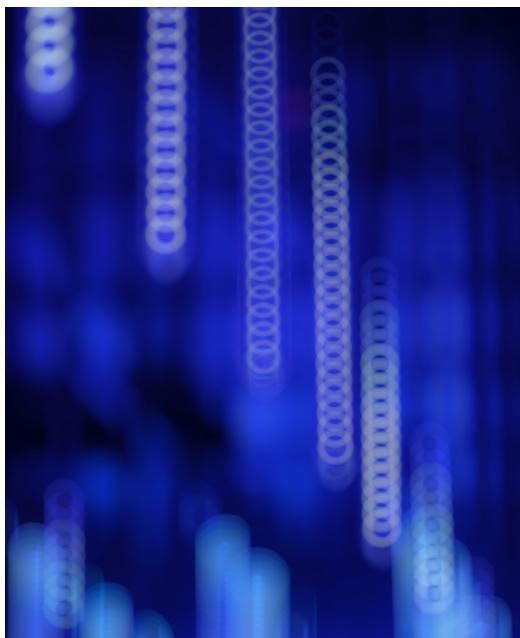
Página oficial en MDN

<https://developer.mozilla.org/es/docs/Web/API/Window>

El objeto “**window**” está soportado en todos los navegadores. Representa la propia ventana del navegador y posee además, algunos métodos integrados:



```
window.open()      // Abre una nueva ventana
window.close()    // Cierra la ventana actual
window.moveTo()   // Mueve a la ventana actual
window.resizeTo() // Reescala la ventana actual
window.location  // Redirige automáticamente a otra ubicación
```



JAVASCRIPT

Módulo XXXII–

DOM

Página oficial en MDN

[https://developer.mozilla.org/es/docs/
Web/API/Document](https://developer.mozilla.org/es/docs/Web/API/Document)

Con el DOM (*Document Object Model*) de HTML, JavaScript puede acceder y realizar cambios en todos los elementos de un documento HTML.

A las acciones que el DOM puede realizar sobre los elementos de HTML se les llama métodos y a los valores que pueden tomar, propiedades.

Al tener acceso a dichos métodos y propiedades, el DOM puede especificarlos o cambiarlos con total libertad.

Selección de elementos

Cuando seleccionamos elementos, guardamos en una variable un elemento de HTML.

Si guardamos varios elementos, entonces se genera una Lista de Nodos



```
// Seleccionamos un elemento id="rojo"
const x = document.getElementById('rojo')
// Seleccionamos un elemento class="cuadrados"
const x = document.getElementsByClassName('cuadrados')
// Seleccionamos un elemento <div>
const x = document.getElementsByTagName('div')
// Seleccionamos un elemento <section>
const x = document.querySelector('section')
// Seleccionamos todos los elementos <section>
const x = document.querySelectorAll('section')
```

Clases

La propiedad **classList** nos permite ejecutar acciones con las clases de un elemento.



```
// Añadimos clases
elemento.classList.add('ver')
// Eliminamos clases
elemento.classList.remove('ver')
// Añadir/Eliminar clases
elemento.classList.toggle('ver')
// Preguntar si existe una clase
elemento.classList.contains('ver')
```

Eventos

Los eventos son acciones que les ocurren a los elementos HTML. Cuando se usa JavaScript en las páginas HTML, éste puede “reaccionar” a esos eventos.

ASIGNAR EVENTOS

Podemos añadir eventos a una o varias etiquetas HTML.



```
function saludo() { console.log(`Evento lanzado`) }

elemento.addEventListener( 'click' , saludo )
elemento.addEventListener( 'click' , function(){ saludo() })
```

TIPOS DE EVENTOS

Existen múltiples tipos de eventos, dependiendo de la acción que busquemos ejecutar.



```
elemento.addEventListener( 'click' , ()=>{} )
elemento.addEventListener( 'dblclick' , ()=>{} )
elemento.addEventListener( 'mouseover' , ()=>{} )
elemento.addEventListener( 'submit' , ()=>{} )
elemento.addEventListener( 'touchmove' , ()=>{} )
```

Crear HTML desde JS

Cuando creamos elementos siempre realizaremos los siguientes pasos:

- Usar **createElement()** dentro de una variable
- Asignar los atributos necesarios
- Añadir esa variable a algún elemento HTML



```
let fragmento = document.createDocumentFragment()

let div      = document.createElement('div')
div.classList.add('cuadrado')
div.innerHTML = 'Esto es el texto'

fragmento.appendChild( div )

document.body.appendChild( fragmento ) // Añade div al final del <body>
document.body.insertBefore( fragmento ) // Añade div al comienzo del <body>
```



JAVASCRIPT

Módulo XXXIII–

JSON

Página oficial en MDN

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON

JavaScript Object Notation o JSON, es un tipo de sintaxis específica de JavaScript para almacenar e intercambiar datos.

De por sí, no existen arrays asociativos en JavaScript, ya que a este tipo de arrays en JavaScript se les conoce como Objetos, concretamente objetos JSON.

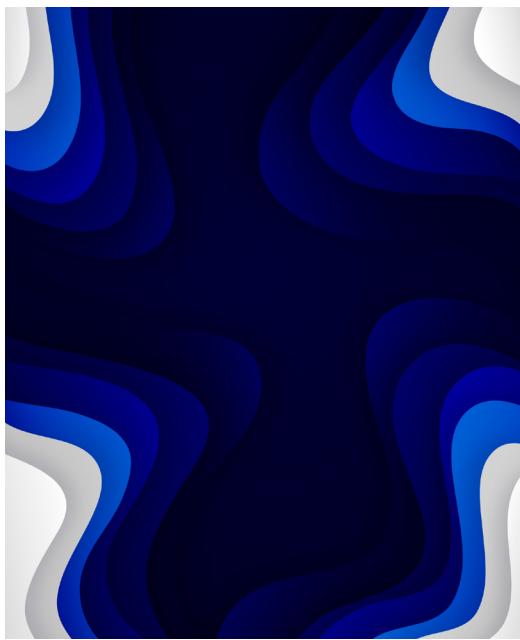
```
{  // Pueden comenzar con llaves o corchetes
  nombre : "Edu",      // Usamos sintaxis de key: value / propiedad:valor
  email  : null,
  vive   : {
    ciudad : "Cádiz",
    cp     : 28001,
  },
  alumnos : [
    { nombre : "Koalba" },
    { nombre : "Ana" },
    { nombre : "Pablo" },
  ]
} // Pueden acabar con llaves o corchetes
```

Si poseemos datos almacenados en un objeto, podemos convertir dicho objeto en JSON y, por ejemplo, enviar sus datos al servidor. Ejemplo:

```
● ● ●  
let myObj      = {nombre: 'Pepe', edad: 30, ciudad: 'Sevilla'}  
let myJSON     = JSON.stringify(myObj)  
window.location = 'demo_json.php?valor=' + myJSON
```

Por otro lado, si queremos recibir datos en formato JSON, también podemos convertir dichos datos en un objeto JSON.

```
● ● ●  
let myJSON      = '{"nombre":"Pepe", "edad":30, "ciudad":"Sevilla"}'  
let myObj       = JSON.parse(myJSON)  
document.querySelector("div").innerHTML = myObj.nombre
```



JAVASCRIPT

Módulo XXXIV–

Fetch

Página oficial en MDN

[https://developer.mozilla.org/es/docs/
Web/API/Fetch_API](https://developer.mozilla.org/es/docs/Web/API/Fetch_API)

Ahora que conocemos JSON vamos al funcionamiento de una petición y cada una de sus partes como:

- Request
- Response
- Promesas
- Fetch

Request

Es un objeto de JavaScript usado para crear una petición de recursos (datos) hacia una URL (normalmente una API).



```
let opciones = {  
    method : 'GET',  
    mode   : 'cors',  
    cache  : 'default',  
    headers: { 'Content-type' : 'application/json' }  
}  
  
let peticion = new Request(`url` , opciones)  
console.log( peticion )
```

Response

Es un objeto de JavaScript que genera una respuesta. El valor de la respuesta al convertirlo usando el método `json()` genera una promesa.



```
// Configuramos las opciones de envío  
let opciones = { "status" : 200 , "statusText" : "¡Suscríbete!" }  
// Configuramos la respuesta  
let miRespuesta = new Response(`{"nombre":"Timmy"}`,opciones);  
  
// Convertimos el String en JSON con .json()  
miRespuesta.json()
```

Promesas

Es un valor con métodos asociados el cual nos *promete* que recibiremos un valor en algún momento.



```
let promesa = new Promise( ( resolve , reject )=> resolve('0') )  
/* Recibe el Response de la promesa */  
.then ( data => data )  
/* Se ejecuta cuando se lanza reject */  
.catch ( data => console.log('Fallo') )  
/* Se ejecuta siempre al acabar */  
.finally( data => console.log('Final') )
```

Fetch

Su uso más habitual es para enviar/recibir los datos de una API externa mediante una *Request* y recibiendo una *Response*.

```
fetch('url')
// Recibe el Response y lo convertimos a JSON
.then ( response => response.json() )
// Aquí ya tenemos los datos en JSON
.then ( data => console.log( data ) )
// Se ejecuta cuando se lanza reject
.catch ( data => console.log('Fallo') )
// Se ejecuta siempre al acabar
.finally( data => console.log('Final') )
```

OPTIONS

Podemos enviar datos extra, como un controlador para cancelar la petición, el tipo de método o los datos extras como los *headers*.

```
let options = {
  method : 'GET', // Configuramos las opciones de conexión
  mode   : 'cors', // Método de petición
  signal : controller.signal, // Modo de petición
  headers: {}, // AbortController
  body   : JSON.stringify(objeto), // Cabeceras
  body   : new FormData() // Cuerpo de la petición
}

fetch('url' , options) // Las aplicamos a la petición
```

CANCELANDO UNA PETICIÓN

El objeto **AbortController** nos permite definir cuándo cancelamos la petición para no dejar la conexión abierta.

```
● ● ●

let controller = new AbortController()           // Creamos un controller
let options    = { signal : controller.signal } // Configuramos signal

let { signal } = new AbortController()           // Con deconstrucción
let options    = { signal }                     // Configuramos signal

fetch('url' , options )                        // Lo añadimos al fetch
```



JAVASCRIPT

Módulo XXXV–

Storage

Página oficial en MDN

<https://developer.mozilla.org/es/docs/Web/API/Storage>

La propiedad **sessionStorage** permite acceder a un objeto Storage asociado a la sesión actual.

La propiedad **sessionStorage** es similar a **localStorage**, la única diferencia es que la información almacenada en **localStorage** no posee tiempo de expiración; por el contrario la información almacenada en **sessionStorage** es eliminada al finalizar la sesión de la página. La sesión de la página perdura mientras el navegador se encuentra abierto.

Abrir una página en una nueva pestaña o ventana iniciará una nueva sesión, lo que difiere en la forma en que trabajan las cookies de sesión.

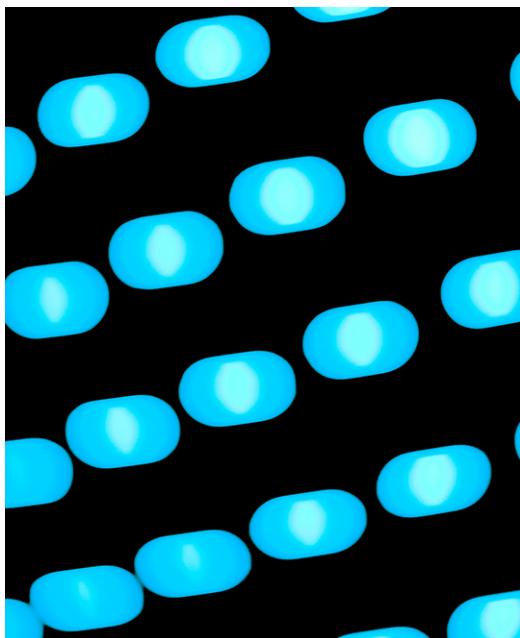
Con **sessionStorage** los datos persisten sólo en la ventana/*tab* que los creó, mientras que con **localStorage** los datos persisten entre ventanas/*tabs* con el mismo origen.

Debe tenerse en cuenta que los datos almacenados tanto en **localStorage** como en **sessionStorage** son específicos del protocolo de la página.

Las claves y los valores son siempre cadenas de texto (ten en cuenta que, al igual que con los objetos, las claves de enteros se convertirán automáticamente en cadenas de texto).



```
localStorage.setItem('nombre', 'valor') // Guardamos un valor  
sessionStorage.setItem('nombre', 'valor') // Guardamos un valor  
  
localStorage.setItem('nombre', 'nuevoValor') // Modificamos el valor  
sessionStorage.setItem('nombre', 'nuevoValor') // Modificamos el valor  
  
localStorage.getItem('nombre') // Obtenemos el valor  
sessionStorage.getItem('nombre') // Obtenemos el valor  
  
localStorage.removeItem('nombre') // Eliminamos el valor  
sessionStorage.removeItem('nombre') // Eliminamos el valor  
  
sessionStorage.clear() // Eliminamos todos los valores  
localStorage.clear() // Eliminamos todos los valores
```



JAVASCRIPT

Módulo XXXVI– *Timers*

Los *timers* en JavaScript son mecanismos que permiten ejecutar código de manera diferida en un momento específico o repetidamente en intervalos definidos. Hay dos tipos principales de *timers* en JavaScript: **setTimeout()** y **setInterval()**.

setTimeout

Página oficial en MDN

<https://developer.mozilla.org/es/docs/Web/API/setTimeout>

Nos permite ejecutar una función de forma retardada, es decir *aplazar* la ejecución de una función definiendo los *milisegundos* que tarda en ejecutarse.

```
// Podemos definir un setTimeout
setTimeout(() => {
    // Función a ejecutar
}, tiempo);

let funcion = () => {}           // Definimos una función
setTimeout( funcion , tiempo ) // La ejecutamos con setTimeout
```

setInterval

Página oficial en MDN

<https://developer.mozilla.org/es/docs/Web/API/setInterval>

Basándonos en la sintaxis de **setTimeout** también existe **setInterval** que nos permite ejecutar una función de forma cíclica (en bucle) midiendo el tiempo en milisegundos.

```
● ● ●

// Creamos una variable para guardar un bucle de tiempo
let auto = setInterval( ()=>{} , tiempo )

// Nos permite limpiar o parar el setInterval
 clearInterval( auto )

// Nos permite volver a ejecutar el setInterval
auto = setInterval( ()=>{} , tiempo )
```

- ★ El índice de este manual es interactivo. Si clicas en cada capítulo o epígrafe te mandará hasta allí. Y si clicas en el numero de página en la esquina inferior izquierda de cada página, volverás al índice.

[volver al índice](#)