



**Centro Integrado de Formación Profesional**  
**AVILÉS**  
Principado de Asturias

## **UNIDAD 4:**

# **MODELO DE OBJETOS DEL DOCUMENTO (DOM)**

**DESARROLLO WEB EN ENTORNO CLIENTE**

**2º CURSO**

**C.F.G.S. DISEÑO DE APLICACIONES WEB**

## REGISTRO DE CAMBIOS

Versión	Fecha	Estado	Resumen de cambios
1.0	05/11/2024	Aprobado	Primera versión
1.1	28/11/2024	Aprobado	Incluidos Web Components
1.2	02/12/2024	Aprobado	Eliminadas referencias obsoletas en jQuery
2.0	21/10/2025	Aprobado	Nueva versión. Eliminadas alternativas de código en jQuery

## ÍNDICE

ÍNDICE .....	1
UNIDAD 4: MODELO DE OBJETOS DEL DOCUMENTO (DOM) .....	2
4.1 El modelo de objetos del documento (Document Object Model - DOM) .....	2
4.1.1 Representación de la página Web como una estructura en árbol .....	2
4.1.2 Diferencias en las implementaciones del modelo .....	2
4.1.3 Objetos del modelo. Propiedades y métodos de los objetos. ....	3
4.2 Manejando el DOM .....	6
4.2.1 jQuery .....	6
4.2.2 Acceso al documento desde código .....	6
4.2.3 Navegando por el DOM .....	9
4.2.4 Modificación dinámica de contenidos y aspecto de una página .....	11
4.2.5 Trabajando con CSS .....	13
4.3 El modelo de eventos .....	15
4.3.1 Programación de eventos. Manejadores de evento más habituales .....	15
4.3.2 Manejadores de evento dinámicos .....	17
4.3.3 Delegación y elementos dinámicos .....	18
4.3.4 Disparando eventos .....	19
4.3.7 Almacenamiento de datos en el cliente .....	20
4.4 Desarrollo de aplicaciones multicliente .....	22
4.4.1 DOM en Aplicaciones multicliente .....	22
4.4.2 Eventos del DOM en distintos navegadores .....	22
4.5 Independencia de las capas de implementación de aplicaciones web .....	24
4.5.1 Web Components .....	26
ÍNDICE DE FIGURAS .....	34
BIBLIOGRAFÍA - WEBGRAFÍA .....	34

## UNIDAD 4: MODELO DE OBJETOS DEL DOCUMENTO (DOM)

### 4.1 EL MODELO DE OBJETOS DEL DOCUMENTO (DOCUMENT OBJECT MODEL - DOM)

El DOM es una API empleada para manipular documentos HTML y XML encargada de extraer los contenidos del documento de texto de forma que aparezcan como estructura jerárquica de objetos. Existen tres especificaciones propuestas por el W3C:

DOM nivel 1: Consiste en dos módulos DOM Core y DOM HTML.

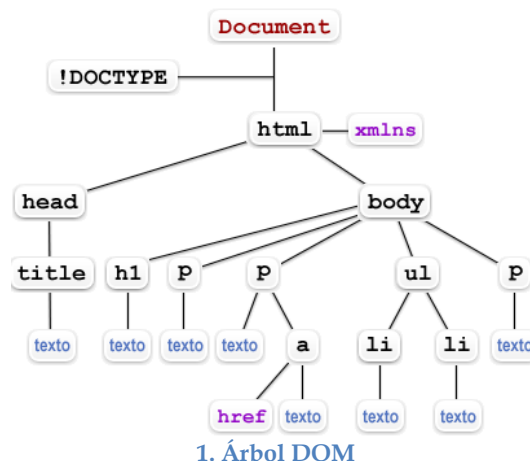
DOM nivel 2: Construido sobre el nivel 1 contiene 14 módulos. Entre los más importantes, está el manejo de eventos de usuario y el de hojas de estilo.

DOM nivel 3: Construido sobre el nivel 2 contiene 16 módulos. Entre los más importantes, el que permite cargar y guardar documentos HTML y XML y el que permite validarlos.

DOM Living Standard: Especificación moderna y en constante evolución del DOM, desarrollada y mantenida por el WHATWG (Web Hypertext Application Technology Working Group). A diferencia de las versiones DOM Level 1, DOM Level 2, etc. que se actualizan de forma independiente el DOM Living Standard es una norma "viva", lo que significa que se actualiza continuamente a medida que se identifican mejoras, correcciones o nuevas necesidades.

#### 4.1.1 REPRESENTACIÓN DE LA PÁGINA WEB COMO UNA ESTRUCTURA EN ÁRBOL

DOM estructura el documento como un árbol jerárquico, tal como se puede ver en la siguiente figura:



DOM trata este árbol como una serie de elementos con sus propiedades, y que la relación entre todos ellos es una relación estructural. Así, por ejemplo, se puede pedir al navegador que devuelva una lista de los hijos del elemento ul.

#### 4.1.2 DIFERENCIAS EN LAS IMPLEMENTACIONES DEL MODELO

Aunque el DOM es un estándar definido por el **W3C (World Wide Web Consortium)**, cada navegador tiene su propia implementación de este modelo a través de su motor de

renderizado. Esas diferencias se han ido minimizando gracias a que el W3C ha promovido una versión estandarizada del modelo ayudando a minimizar estas diferencias.

Históricamente, cada uno de los navegadores implementaban el DOM a su manera haciendo que fuera más complicada y farragosa la tarea del desarrollo en cliente. Por ejemplo, Microsoft implementaba su propia versión del DOM llamada “**proprietary DOM**” para **Internet Explorer** la cual difería del estándar propuesto por el W3C. Otro mecanismo que se usaba anteriormente consistía en implementar métodos con prefijos propios del motor de renderizado; por ejemplo, **window.webkitRequestAnimationFrame** (prefijo webkit en referencia al motor de renderizado del navegador Safari) o **mozRequestFullScreen** (prefijo moz de Mozilla para Firefox).

En el contexto de navegadores modernos, algunos pueden tardar más en adoptar nuevas APIs del DOM o hacerlo de forma distinta, aunque gracias a los esfuerzos de estandarización, las diferencias entre las implementaciones del DOM ahora mismo son mucho menores que en el pasado.

#### 4.1.3 OBJETOS DEL MODELO. PROPIEDADES Y MÉTODOS DE LOS OBJETOS.

Los nodos representan los diferentes niveles de los documentos HTML y XML. Existen nodos de diferente tipo (**Document, Document Type, Element, Attribute, Text, Comment**) y diferente nivel.

Tipos de nodos en el DOM: descripción y posibilidad de tener hijos o no		
Nodo	Descripción	¿Hijos?
Element.Node	Representa el contenido definido desde una etiqueta de apertura a una de cierre	Sí
Attribute_Node	Representa un atributo o propiedad de un elemento.	No
Text_Node	Representa/almacena el contenido del texto o datos	No
CData_Section_Node	Representa una sección <b>&lt;![CDATA[]]&gt;</b> .	Sí
Entity_Reference_Node	Representa una referencia de entidad.	No
Entity_Node	Representa la definición de una entidad en la <b>DTD</b> .	No
Processing_Instruction_Node	Representa una instrucción de proceso.	No
Comment_Node	Utilizado para indicar los comentarios producidos en el documento.	No
Document_Node	Nodo raíz de los documentos HTML/XML. Solo hay uno por documento DOM	Sí
Document_Type_Node	Contiene el DTD empleado en el documento. También llamado DOCTYPE. Informa cual es la estructura o la versión de HTML que se va a usar para validar el documento	No
Document_Fragment_Node	Representa un fragmento del modelo total del árbol del documento.	Sí
Notation_Node	Representa una notación definida en la <b>DTD</b> .	No

Existe un objeto general **NODE** que permite obtener información sobre los nodos y manipular su contenido. Este objeto define una serie de propiedades y métodos, así como constantes. Las constantes permiten identificar el tipo de nodo.

- `Node.ELEMENT_NODE = 1`
- `Node.ATTRIBUTE_NODE = 2`
- `Node.TEXT_NODE = 3`
- `Node.CDATA_SECTION_NODE = 4`
- `Node.ENTITY_REFERENCE_NODE = 5`
- `Node.ENTITY_NODE = 6`
- `Node.PROCESSING_INSTRUCTION_NODE = 7`
- `Node.COMMENT_NODE = 8`
- `Node.DOCUMENT_NODE = 9`
- `Node.DOCUMENT_TYPE_NODE = 10`
- `Node.DOCUMENT_FRAGMENT_NODE = 11`
- `Node.NOTATION_NODE = 12`

Existen navegadores que no soportan las constantes predeterminadas. Si es así, hay que definirlas de la siguiente forma:

```
if(typeof Node == "undefined") {  
    let Node = { ELEMENT_NODE: 1, ATTRIBUTE_NODE: 2,  
        TEXT_NODE: 3, CDATA_SECTION_NODE: 4,  
        ENTITY_REFERENCE_NODE: 5, ENTITY_NODE: 6,  
        PROCESSING_INSTRUCTION_NODE: 7, COMMENT_NODE: 8,  
        DOCUMENT_NODE: 9, DOCUMENT_TYPE_NODE: 10,  
        DOCUMENT_FRAGMENT_NODE: 11, NOTATION_NODE: 12};  
}
```

Para gestionar el objeto **NODE** se dispone de una serie de propiedades que permitan acceder a la información o características de los nodos elegidos incluso modificándolo en algunos casos.

Atributos	Definición
<code>nodeName</code>	El nombre del nodo
<code>nodeValue</code>	El valor del nodo
<code>nodeType</code>	Un código que representa el tipo del objeto subyacente
<code>ownerDocument</code>	Objeto Document asociado a ese nodo
<code>firstChild</code>	El primer hijo de este nodo. Si no existe, devuelve null
<code>lastChild</code>	El último hijo. Si no existe, devuelve null
<code>childNodes</code>	Lista NodeList que contiene todos los hijos de este nodo
<code>previousSibling</code>	Nodo inmediatamente precedente a este. Si no existe, devuelve null
<code>nextSibling</code>	Nodo que sigue inmediatamente al actual. Si no existe, devuelve null
<code>attributes</code>	Atributos del nodo

De la misma forma, en el objeto NODE se dispone de una serie de métodos que permiten interactuar con los nodos elegidos consultando y modificando su estructura.

hasChildNodes()	Permite determinar si un nodo tiene algún hijo
appendChild(nodo)	Añade el nodo al final de la lista de hijos de este nodo. Si ya está en el árbol, se elimina y se vuelve a crear
removeChild(nodo)	Retira el nodo hijo indicado en la lista de hijos, y lo devuelve como valor de retorno
replaceChild(nuevo,anterior)	Reemplaza el nodo hijo anterior con el nuevo y devuelve el anterior. Si el nuevo ya está en el árbol, se reemplaza
insertBefore(nuevo, anterior)	Inserta el nodo nuevo antes del existente. Si el anterior es null, se inserta al final de la lista de hijos

## 4.2 MANEJANDO EL DOM

En este apartado se verá cómo navegar sobre el DOM, localizar elementos y modificarlos para el manejo de este modelo.

### 4.2.1 JQUERY

jQuery es una biblioteca que facilita la tarea de acceder al DOM mediante una sintaxis más amigable a priori. Para poder utilizarlo, una de las opciones consiste en instalarse la versión comprimida y de producción de jQuery (última versión a la fecha de este texto, la 3.7.1 de agosto de 2023). Se puede descargar en el siguiente enlace accediendo al botón Download jQuery 3.7.1: <https://jquery.com/download/>

En el directorio de ejemplo, el .js bajado (jquery-3.x.y.z.min.js) se coloca en el llamado Scripts, como cualquier otro fichero JavaScript. Se puede probar con este script básico:

```
<html>
  <head>
    <title>Hola, jQuery!</title>
  </head>
  <body>
    <script src='./Scripts/jquery-3.7.1.min.js'></script>
  </body>
</html>
```

Para referenciar a jQuery se puede usar un CDN (Content Delivery Network). Lo que garantiza el CDN es que se va a tener el contenido del **framework jQuery** fuera del servidor local, pero que el CDN se encargará de proporcionar el archivo en el servidor más cercano y rápido al cliente que haga uso de la aplicación, además de ser bastante probable que el navegador cachee su contenido. Basta con hacer una búsqueda de un CDN que aloje **jQuery** (existen similares para Bootstrap, Angular, etc.) y se encontrarán unos cuantos. En este ejemplo se probará con el de Microsoft, cuya dirección en el momento de elaborar este documento es la siguiente:

<https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.7.1.js>

Por tanto, se cambiará el origen de jQuery para adaptarlo a este CDN:

```
<script src='https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.7.1.js'></script>
```

OJO: Aunque en este caso puede tentar cerrar la etiqueta de script sin su equivalente de cierre (simplemente con />), si se opera así, la etiqueta no funcionará correctamente, por lo que es esencial mantener la sintaxis mostrada en estos ejemplos.

### 4.2.2 ACCESO AL DOCUMENTO DESDE CÓDIGO

A continuación, se muestran algunos ejemplos de acceso al documento desde código JavaScript.

Por ejemplo, si se quiere obtener el documento completo desde las etiquetas `<html>` a `</html>`

```
const objetoCompleto = document.documentElement;
```

Cada nodo tiene una propiedad **firstChild** que apunta al nodo del que forma parte en caso de que exista. Del mismo modo, cada nodo elemento tiene una propiedad **childNodes** que apunta a un array que contiene sus hijos. De esta forma, es posible desplazarse a cualquier parte del árbol usando solo estos enlaces padre e hijo. Además, JavaScript también permite acceder a una serie de enlaces adicionales. Las propiedades **firstChild** y **lastChild** apuntan al primer y último elemento hijo o valen **null** en caso de nodos sin hijos. Con el siguiente código se obtiene la cabecera (**head**) y el cuerpo (**body**):

```
let objetoCabecera = objetoCompleto.firstChild;
let objetoCuerpo = objetoCompleto.lastChild;

let objetoCabecera=objetoCompleto.childNodes[0];
let objetoCuerpo=objetoCompleto.childNodes[1];
```

El tipo de nodo, devuelto como number, se obtiene de la forma siguiente:

```
let tipoNodo;
tipoNodo = document.nodeType;
tipoNodo = objetoCabecera.nodeType;
tipoNodo = document.documentElement.nodeType;
```

El siguiente código proporciona el número de descendientes de un objeto:

```
const descendientes = objetoCompleto.childNodes.length;
```

Cada nodo de tipo **element** contiene una colección **NamedNodeMap** para acceder a los atributos de cada uno de los elementos. Se puede obtener un nodo de tipo **Attribute** mediante diversos métodos:

- .getNamedItem(valor):** Contiene el nodo cuyo nombre es idéntico al valor pasado
- .setNamedItem(valor):** Añade un nuevo nodo al documento con el valor pasado
- .removeNamedItem(valor):** Elimina el nodo indicado en el valor pasado
- .item(posicion):** Devuelve el nodo que se encuentra en la posición indicada

Supóngase el siguiente código:

```
<p id= "Ejemplo">Ejemplo de prueba</p>
const elemento=document.getElementById("Ejemplo");
```

Con él se obtiene como elemento el objeto con identificador Ejemplo.

```
let valor=elemento.attributes.getNamedItem("id").nodeValue;
```

Aquí valor contendrá Ejemplo:

```
elemento.setAttribute("id", "cambio");
```

En este caso, se cambia el nombre del id por cambio. Después de realizar estos cambios, se tendría:

```
<p id="cambio">Ejemplo de prueba </p>
```



Métodos en DOM:

**.createAttribute(Nombre):** Crea un nodo del tipo attribute  
**.createTextNode(texto):** Crea un nodo de tipo text  
**.createElement(tipo):** Crea un elemento del tipo indicado  
**.createComment(texto):** Crea un nodo de tipo comment  
**.appendChild(valor/objeto):** Añade el valor o un objeto a continuación dentro del documento  
**.insertBefore(objeto\_nuevo, objeto\_existente):** Añade el objeto delante del objeto indicado  
**.removeChild(nodo):** Borra cualquier nodo original o creado dinámicamente  
**.replaceChild(nodo):** Modifica el contenido que existe en el nodo indicado  
**.getElementsByName(etiqueta):** Obtiene todos los elementos de la página cuya etiqueta se igual al parámetro pasado. Devuelve un array (NodeList)  
**.getElementsByName(nombre):** Obtiene todos los elementos de la página cuyo nombre sea igual al parámetro pasado. Normalmente solo uno. Devuelve Array(nodelist)  
**.getElementById(id):** Se usa para llegar directamente a un elemento. Devuelve un elemento HTML cuyo atributo **id** coincide con el **parámetro** dado.

#### 4.2.2.1 Gestión de enlaces de un documento

En este ejemplo, se puede ver cómo gestionar los enlaces de un documento. La secuencia de código en la sección HTML <body> del script:

```
<body>
<!-- Definición de varios enlaces de hipertexto consecutivos en el documento -->
Motor de búsqueda <a href="http://www.google.es" id="google">Google</a>
<br>
Sitio de información <a href="http://www.yahoo.es" id="yahoo">Yahoo</a>
<br>
Enciclopedia <a href="http://www.wikipedia.es" id="wikipedia">Wikipedia</a>
<script>
  document.write("<br><br>");
  // Se muestra el número de enlaces del documento
  document.write("Número de enlace(s) del documento: ",document.links.length,"<br>");
  // Se muestra el identificador del segundo enlace del documento */
  document.write("Id del segundo enlace del documento:",document.links[1].id,"<br>");
  /* Se muestra la URL del tercer enlace del documento */
  document.write("URL del tercer enlace del documento: ",document.links[2].href);
</script>
</body>
```

Se incluyen tres enlaces (google, yahoo y wikipedia) con etiquetas HTML <a href> ... </a>. Posteriormente, se muestra el número de enlaces del documento, el identificador del segundo enlace y la URL del tercer enlace. Hay que tener en cuenta que la numeración de los enlaces se realiza de 0 a n-1.

#### 4.2.2.2 Gestión de imágenes de un documento

En este apartado se verá cómo gestionar imágenes mediante un ejemplo. La secuencia de código es la siguiente:

```
<!-- Definición de varias imágenes consecutivas en el documento-->

<br><br>

<br><br>

<br><br>

<br><br>

<script>
/* Se muestra el número de imágenes del documento */
document.write("<br>Número de imágenes del documento: ", document.images.length);
/* Se muestra el identificador de la primera imagen */
document.write("<br>Identificador de la 1ª imagen: ", document.images[0].id);
/* Se muestra el atributo border de la segunda imagen */
document.write("<br>Atributo border de la 2ª imagen: ", document.images[1].border);
/* Se muestra el atributo src de la tercera imagen */
document.write("<br>Atributo src de la 3ª imagen: ", document.images[2].src);
/* Se muestran los atributos width y height de la cuarta imagen */
document.write("<br>Atributo width de la 4ª imagen: ", document.images[3].width);
document.write("<br>Atributo height de la 4ª imagen: ", document.images[3].height);
</script>
```

Se presentan cuatro imágenes con determinados atributos (id, border, src, width, height). Posteriormente, se muestra diferente información respecto a estas imágenes como el número de ellas en el documento y alguno de sus atributos.

### 4.2.3 NAVEGANDO POR EL DOM

#### 4.2.3.1 Accediendo a elementos

Como se vio anteriormente, gracias a propiedades como **firstChild**, **lastChild**, **nextSibling**, etc. es posible moverse por el modelo. Además de ellos, existe una propiedad **children** que es equivalente a **childNodes** pero solo contiene hijos que son elementos, no de otro tipo. Esto puede ser útil cuando no se está interesado en nodos de texto.

Cuando se trabaja con una estructura de datos anidada como esta, las funciones recursivas suelen ser muy útiles.

La siguiente función explora un documento en busca de nodos de texto que contengan una cadena dada y devuelve **true** cuando encuentra uno.

```
const documentoContiene = (nodo, cadena) => {  
  if (nodo.nodeType == Node.ELEMENT_NODE) {  
    for (let i = 0; i < nodo.childNodes.length; i++) {  
      if (documentoContiene(nodo.childNodes[i], cadena)) {  
        return true;  
      }  
    }  
    return false;  
  } else if (nodo.nodeType == Node.TEXT_NODE)  
    return nodo.nodeValue.indexOf(cadena) > -1;  
};  
console.log(documentoContiene(document.body, "patrón de búsqueda"))
```

La llamada a la función se hace sobre el **body** del documento. Es preciso tener en cuenta que si se hace la llamada al método desde el propio elemento **body**, siempre devolverá true al ir incluido el elemento en el código JavaScript.

Navegar por enlaces entre padres, hijos y hermanos puede ser útil pero no siempre recomendable ya que el programa debería hacer suposiciones sobre la estructura del documento. Otro factor que complica las cosas es que los nodos de texto se crean incluso para el espacio en blanco entre nodos; es decir, una etiqueta <body> no solo tendrá otros elementos (h1, p, etc.) sino también los espacios en blanco antes y después de ellos. Por ejemplo, dada esta página:

```
<!DOCTYPE html>  
<html>  
  <head>  
    <title>Mi página personal</title>  
  </head>  
  <body>  
    <h1>Mi página personal</h1>  
    <p>Hola, me llamo Jason y esta es mi página principal.</p>  
    <p>  
      Además, escribo libros; no sé a qué esperas para leerlos  
      <a href="http://mislibricos.net">aquí</a>  
    </p>  
  </body>  
</html>
```

Si antes se vio cómo poder acceder a los enlaces de un documento, en este caso se podría querer hacer de otro modo, por ejemplo, obteniendo el atributo **href**. Tal como se hizo anteriormente sería algo como “quiero el segundo hijo del cuerpo del documento”. Obviamente, la estructura del documento puede cambiar y esa forma tan rígida dejar de ser válida, así que la mejor opción sería “quiero el primer enlace del documento”.

Se haría así:

```
const enlace = document.body.getElementsByTagName("a")[0];  
console.log(enlace.href);
```

En definitiva, todos los nodos de elemento tienen un método **getElementsByTagName** que recoge todos los elementos con el nombre de etiqueta dado que son descendientes (sean directos o indirectos) de ese nodo y los devuelve como un objeto con forma de array.

Si se quiere encontrar un nodo individual, se le puede dar un atributo id y utilizar **document.getElementById**:

```
<p>Mi imagen de marca</p>  
<p><img id='imagenMarca' src='img/imagenMarca.png'></p>  
<script>  
    const imagen = document.getElementById('imagenMarca');  
    console.log(imagen.src);  
</script>
```

#### 4.2.4 MODIFICACIÓN DINÁMICA DE CONTENIDOS Y ASPECTO DE UNA PÁGINA

En este apartado se van a tratar algunas de las funciones que JavaScript ofrece para acceder y reemplazar valores de elementos, atributos y CSS.

##### 4.2.4.1 Recuperación de un valor

Supóngase que se tiene este formulario:

```
<form>  
    <label for="txtEntrada">Introduce tu texto aquí:</label><br><br>  
    <input type="text" id="txtEntrada"/>  
    <input type="button" value="Clic"/>  
    <label id="lblError"></label>  
</form>
```

Para recuperar el valor de ese control input se puede usar el método **getElementById** y, una vez obtenido el elemento, utilizar su propiedad **value**. Mediante esa misma propiedad, se puede modificar el valor. Por ejemplo, si se quiere limpiar el cuadro de texto (por ejemplo, con la acción de un botón), se haría lo siguiente:

```
document.getElementById("txtEntrada").value = "";
```

##### 4.2.4.2 Añadiendo y eliminando clases en un objeto

Si se tienen una serie de clases con estilos predefinidos, se pueden añadir dinámicamente a los objetos de una página o también eliminarlas.

Supóngase la siguiente clase que cambia el color de fondo de un control por azul y el del texto por blanco:

```
<style>
  .blue { background-color: blue; color: white; }
</style>
```

Para añadir a un control el estilo dado por esa clase se hace lo siguiente:

```
const elemento = document.getElementById("target");
elemento.classList.add("blue");
```

#### 4.2.4.3 Contenido de un elemento

En JavaScript si se quiere obtener el texto de un contenido se haría así:

```
document.getElementById("output").textContent = "valor";
```

Para obtener el texto visualizado como HTML se utilizaría este código:

```
document.getElementById("output").innerHTML = "<b>valor</b>";
```

#### 4.2.4.4 Añadiendo nuevos elementos

Si se quiere añadir un nuevo contenido a un elemento, el cual puede contener otros nuevos, existen dos opciones: hacerlo al final del elemento objetivo o al principio.

Supóngase el siguiente código:

```
<div id="target">
  <div>Contenido molón</div>
  <div>Contenido súper molón</div>
</div>
```

Para añadir un contenido al final en JavaScript, se usa el método **appendChild**. Si se quiere añadir al final de un elemento target se haría lo siguiente:

```
const target = document.getElementById('target');
let nuevoContenido = document.createTextNode("Texto agregado");
target.appendChild(nuevoContenido);
```

Si se quiere añadir contenido al comienzo de un elemento en JavaScript:

```
target.insertBefore(nuevoContenido, target.firstChild);
```

En JavaScript hay que hacer referencia al nodo padre del objetivo e insertarlo antes del próximo hermano del objetivo:

```
target.parentNode.insertBefore(nuevoContenido, target.nextSibling);
```

Para añadir un elemento antes de un contenido, hay que hacerlo directamente sobre aquel y no sobre un hermano:

```
target.parentNode.insertBefore(nuevoContenido, target);
```

#### 4.2.4.5 Eliminando y sustituyendo elementos

Para eliminar un elemento del DOM en JavaScript, hay que recurrir a `removeChild` desde el nodo padre.

```
target.parentNode.removeChild(target);
```

Esta orden elimina completamente el elemento `target` del DOM. En el ejemplo anterior eliminaría la `div` etiquetada como `target` al completo. Pero ¿y si lo que se desea es limpiar el contenido de esa `div`? Pues algo tan simple como vaciar su HTML interno:

```
target.innerHTML = "";
```

JavaScript tiene también un método `replaceWith`, pero OJO, el código anterior sobrescribiría el contenido con el texto literal, sin interpretar HTML, es decir, solo funciona si se ha creado un elemento, por ejemplo, con `createElement`:

```
let nuevoContenido = document.createElement("div");  
nuevoContenido.innerHTML = "NUEVO contenido";  
target.replaceWith(nuevoContenido)
```

#### **4.2.5 TRABAJANDO CON CSS**

Aunque se puede cambiar el estilo de un elemento a través de clases, podría ser necesario modificar el CSS de un solo elemento directamente. En este caso se puede modificar el atributo `style`:

```
document.getElementById('target').setAttribute('style', 'color: red');
```

Esto podría funcionar, pero no es ideal. En este ejemplo, se va a reemplazar el atributo `style` de forma que el resto de los estilos se perderían (en caso de que los hubiera). En JavaScript sería mejor utilizar el siguiente código:

```
document.getElementById('target').style.color = 'red';
```

con lo que solo se modifica el color del estilo sin sobrescribir el resto de los estilos.

En JavaScript se puede obtener el color de un elemento de esta forma:

```
let color = document.getElementById('target').style.color;
```

#### 4.2.5.1 Trabajando con selectores

En JavaScript se puede localizar cada elemento con su estilo predeterminado:

```
<h1>Hola, mundo</h1>  
document.querySelector('h1');
```

Pero si se quiere hacer de forma más precisa, se puede seleccionar un conjunto de elementos por su clase CSS:

```
<h1 class="centered">Hola, mundo</h1>  
document.querySelectorAll('.centered');
```

Se puede modificar el texto de todos los elementos **h1** de una página. Igualmente, con aquellos de una clase concreta, y por supuesto, de un **id**:

```
document.querySelectorAll("h1").forEach(  
    (element) => element.textContent = "Hola, JavaScript");
```

#### 4.2.5.2 Selectores basados en atributos

Con JavaScript también se pueden localizar elementos basados en un valor de un atributo. Por ejemplo:

```
<h1 custom="banner">Hola, mundo</h1>  
// En JavaScript  
document.querySelectorAll('h1[custom="banner"]');
```

Es decir, usando corchetes se puede localizar un elemento con un atributo de un valor concreto. Supóngase este otro elemento:

```
<h1 class="col-md-5">Hola, mundo</h1>
```

En el caso de una clase CSS, se podía haber hecho con **h1.col-md-5**

Si lo que se quiere es referenciar a todos los elementos **h1** de cualquier clase que empiece por “col” hay que usar el siguiente selector:

```
'h1[class^="col"]'
```

En este caso se usa el acento circunflejo ^, carácter comodín que corresponde a “empieza por”. Si lo que se desea es que contenga cualquier valor (por ejemplo “md”) habría que usar un asterisco:

```
'h1[class*="md"]'
```

#### 4.2.5.3 Selectores posicionales

Se pueden buscar relaciones de tipo padre/hijo entre elementos como en el ejemplo a continuación:

```
<nav>  
    <a href="#">Vínculo</a>  
</nav>
```

Y referenciar al vínculo de esta forma:

```
'nav > a'
```

lo que haría sería buscar todos aquellos enlaces que sean hijos de un elemento de navegación **nav**. Ojo, porque este sistema sólo funciona si es un descendiente directo (es decir, de padre a hijo). Si se tuviera una jerarquía mayor, y por poner un ejemplo, se buscasen los elementos bisnietos de **nav** que representan enlaces, habría que hacer lo siguiente:

```
'nav a'
```



### 4.3 EL MODELO DE EVENTOS

Las páginas web se construyen típicamente usando una arquitectura basada en eventos. Un evento en una arquitectura web se puede definir como el mecanismo que se acciona cuando el usuario realiza un cambio sobre la página.

Principalmente, lo que interesa es capturar ese evento, es decir, programar una acción que realice una tarea justo cuando se dispare. Para ello se usa lo que se conoce como manejador de evento, que no es otra cosa que la acción que se va a manejar al lanzarse dicho suceso. Un ejemplo muy simple es el evento que sucede al hacer clic sobre un elemento de la página con el ratón y su manejador se denomina **onClick**. En este apartado se verá cómo gestionar dichos eventos mediante el DOM. La lista completa de eventos que pueden manejarse se puede encontrar en la siguiente URL:

[https://www.w3schools.com/jsref/dom\\_obj\\_event.asp](https://www.w3schools.com/jsref/dom_obj_event.asp)

Se verán posteriormente otros modelos de eventos, aunque en este apartado, el que interesa es el modelo de registros avanzados del W3C. La principal característica de este tipo de modelo de eventos es que separa el etiquetado HTML del código JavaScript.

#### 4.3.1 PROGRAMACIÓN DE EVENTOS. MANEJADORES DE EVENTO MÁS HABITUALES

Supóngase el siguiente código:

```
<body>
  <h1>Modelo de eventos del W3C</h1>
  <h3 id="w3c">Modelo W3C</h3>
  <h3 id="w3canonima">Modelo W3C con funciones anónimas</h3>
  <script>
    const saludarUnaVez = () => {
      alert("Hola, mundo");
    }
    document.getElementById("w3c").addEventListener("click", saludarUnaVez, false);
  </script>
</body>
```

Como se puede ver, primeramente, se selecciona el elemento por su identificador y de una forma muy concreta, se le añade un **EventListener**, lo que viene a ser un “escuchador de eventos” el cual recoge el evento a escuchar (**click**), el método a llamar y el último booleano, el cual indica si los eventos de este tipo se enviarán al **listener** registrado antes de enviarse a cualquier **EventTarget** debajo de él en el árbol DOM. En definitiva, si se va a propagar a otros controles.

Es fácil observar que con hacer clic sobre el elemento **h3** con identificador **w3c** el sistema lanza un mensaje emergente. Pero, además es posible añadir más de un manejador de evento sobre una operación.



Se modificará el elemento anterior para modificar su color a rojo.

```
const saludarUnaVez = () => alert("Hola, mundo");
const colorear = () => (document.getElementById("w3c").style.color = "red");
const w3c = document.getElementById("w3c");
w3c.addEventListener("click", saludarUnaVez, false);
w3c.addEventListener("click", colorear, false);
```

Por supuesto, no solo se puede añadir un evento **click** a un elemento, también cualquier otro de los indicados en el enlace visto anteriormente. En el siguiente ejemplo, se cambia el fondo cuando el puntero del ratón se posiciona sobre el elemento:

```
const fondo = () => document.getElementById("w3c").style.background = "blue";
w3c.addEventListener("mouseover", fondo, false);
```

Si la idea es que cuando se lance el evento **saludarUnaVez** no se vuelva a ejecutar, la operación a realizar consistiría en eliminar dicho manejador de la colección del elemento. Y para eso, no hay más que llamar a **removeEventListener**. Para ello, se modificaría el manejador de esta forma:

```
const saludarUnaVez = () => {
    alert("Hola, mundo");
    document.getElementById("w3c").removeEventListener("click", saludarUnaVez);
};
```

Una vez hecho esto, la ventana emergente no volverá a aparecer.

Pero no es estrictamente necesario que el evento esté asociado a una función con un nombre; otra forma de operar es utilizando funciones anónimas. La operativa es similar, ya que se llama al mismo código, pero en lugar de introducir la función con un nombre, se añade directamente su código con un formato como el que se ve en el siguiente código:

```
document.getElementById("w3canonima").addEventListener("click",
    function () {
        this.style.background = "#C0C0C0";
    });
```

Como puede verse, modifica el color de fondo del elemento a gris. En este caso, no se están usando expresiones lambda para la función anónima porque con este código no funcionaría. Esto se debe a que una función **arrow**, **this** no se refiere al elemento que disparó el evento, a diferencia de la función tradicional **function**. En estas funciones **arrow**, **this** hereda el contexto del ámbito padre (donde se ha definido), por lo que cambia completamente su significado. Para poder usar una función **arrow** anónima, hay que hacer referencia al propio evento y a una de sus propiedades. Aquí se pueden usar tanto **event.target** como **event.currentTarget**. La primera hace referencia al elemento que originó el evento y la segunda se refiere al elemento que contiene el **addEventListener** y sería la opción más ajustada al uso de **this**.

El código quedaría así:

```
document.getElementById("w3canonima").addEventListener(  
    "click", (event) => (event.currentTarget.style.background = "#C0C0C0")  
);
```

Cuando se registra un manejador de evento de una parte diferente del objeto que lanzará el evento, hay una desconexión entre ambos objetos. Si se mantiene una ratio 1:1 entre manejadores y eventos, de forma que cada evento en el que se esté interesado tenga su propia función, generalmente no habrá dudas en saber qué ha ocurrido y sobre qué objeto lo ha hecho. Sin embargo, incluso aun manteniendo esa ratio de 1:1 puede haber una frágil relación. Si algo cambia en el objeto, el código podría no funcionar con los cambios que se hayan hecho. En suma, normalmente se reutilizará un sólo manejador de evento para varios eventos.

Cuando se crean manejadores de evento, a menudo es mejor no asumir nada acerca del objeto que ha lanzado el evento, incluso su identificador. No sólo las cosas pueden cambiar hasta el punto de romper el código, si se evita este vínculo entre el manejador del evento y el objeto, el código puede volverse más reutilizable y flexible.

Cada manejador de evento puede incluir un objeto **event** que va a depender del tipo de elemento disparado (ya se utilizó para las funciones **arrow**). En la siguiente URL se pueden consultar sus propiedades y métodos: [https://www.w3schools.com/jsref/obj\\_event.asp](https://www.w3schools.com/jsref/obj_event.asp) (**Event Properties and Methods**).

Por ejemplo, en caso de eventos de ratón ([Mouse Events](#)), además de las propiedades del objeto base **Event**, se pueden encontrar las siguientes:

- **pageX** y **pageY** para las coordenadas donde está el ratón relativas al documento.
- **button** identifica el botón que ha lanzado el evento.

Para acceder al objeto event, únicamente hay que añadir un parámetro al manejador:

```
(event) => alert(event.pageX);
```

### 4.3.2 MANEJADORES DE EVENTO DINÁMICOS

Hasta ahora se ha utilizado el formato **selector.addHandler('click', function)** para asociar manejadores de eventos.

Ejemplo:

```
<html>
  <head>
    <title>¡Hola, JavaScript!</title>
    <style>
      .btn {
        font-size: 1em;
      }
    </style>
  </head>
  <body>
    <form>
      <div><button class="btn validate">Primer botón</button></div>
      <div><button class="btn validate">Segundo botón</button></div>
      <div><button class="btn">Tercer botón</button></div>
    </form>
    <div id="display"></div>
    <script>
      const form = document.querySelector("form");
      // Se delega el evento click en el formulario
      form.addEventListener("click", (event) => {
        if (event.target.matches("button.validate"))
          document.getElementById("display").textContent = "Clicado!";
      });
    </script>
  </body>
</html>
```

Se recoge el evento desde la función y su propiedad target, que representa al control que lo ha lanzado. Si coincide con **button.validate**, ejecuta el código correspondiente.

### 4.3.3 DELEGACIÓN Y ELEMENTOS DINÁMICOS

Una pregunta que puede surgir es qué va a ocurrir en este escenario:

```
<button>Clic</button>
<div id="placeholder"></div>
<script>
  // Se registra un evento click con el botón existente
  document
    .querySelector("button")
    .addEventListener("click", () => alert("Hola"));
  // Se crea un nuevo botón y se añade al contenedor
  const nuevoBoton = document.createElement("button");
  nuevoBoton.textContent = "Nuevo button";
  document.getElementById("placeholder").appendChild(nuevoBoton);
</script>
```

Cuando el código se ejecuta, se crea un nuevo botón una vez registrado un manejador de evento para todos los botones. La pregunta es si el nuevo botón tendrá el mismo manejador de evento. El manejador se enlaza cuando el código es ejecutado, por lo que el selector usado no es reevaluado. Por tanto, **los objetos añadidos a posterioridad no tendrán el mismo manejador.**

La delegación de eventos solventa este problema ya que permite añadir el manejador para futuros elementos. En el ejemplo anterior:

```
document.addEventListener("click", (event) => {  
    // Verifica si el objetivo del evento es un botón  
    if (event.target.matches("button")) {  
        alert("hola");  
    }  
});
```

Es decir, se añade un evento click para todo el documento permitiendo capturar clics en cualquiera de sus elementos. Dado que se puede obtener el control sobre el que se ha lanzado el evento mediante **event.target** bastaría con evaluar que es un botón para lanzar las acciones correspondientes.

#### 4.3.4 DISPARANDO EVENTOS

En ocasiones se necesita lanzar eventos de forma programática. Para lanzar un evento, simplemente hay que llamar al método de registro del evento sin parámetros:

```
elemento.click();
```

Pero si se quisiera ejecutar el código del manejador sin llegar a ejecutar el evento, se pueden usar uno de estos dos métodos:

- Llamar a los manejadores de evento para todos los elementos coincidentes:

```
const element = document.querySelector('selector');  
// Crea un nuevo evento  
const event = new Event('event');  
// Dispara el evento  
element.dispatchEvent(event);
```

- Llamar al manejador de evento para sólo el primer elemento coincidente, es decir, sin propagación:

```
const element = document.querySelector('selector');  
const event = new Event('eventName', {  
    bubbles: false, // No se propaga  
    cancelable: true // Puede ser cancelado  
});  
element.dispatchEvent(event);
```

### 4.3.7 ALMACENAMIENTO DE DATOS EN EL CLIENTE

Las páginas HTML simples con JavaScript pueden ser un formato excelente para realizar pequeñas aplicaciones que automaticen tareas básicas. Cuando una aplicación de este tipo necesita recordar algo entre sesiones, habría que recurrir a tecnologías de servidor como se verá posteriormente, pero en este caso simplemente se contemplará la conservación de datos locales en el navegador.

El objeto **localStorage** se puede usar para almacenar datos de una manera que sobrevivan a la recarga de páginas. Este objeto permite almacenar valores de cadena bajo un nombre concreto.

```
localStorage.setItem("nombreusuario", "mario");  
console.log(localStorage.getItem("nombreusuario")); // Devolvería Mario  
localStorage.removeItem("nombredeusuario");
```

Un valor en **localStorage** permanece hasta que se sobrescribe, se quita con **removeItem** o cuando el usuario borra sus datos locales. El siguiente código implementa una aplicación sencilla para tomar notas. Mantiene un conjunto de notas con un nombre además de permitir que el usuario las edite y cree otras nuevas.

```
<html><head><title>Documento</title></head>  
<body>  
  <select></select>  
  <button>Añadir nota</button><br />  
  <textarea style="width: 100%"></textarea>  
  <script>  
    // Representa el desplegable de los títulos  
    const cbTitulos = document.querySelector("select");  
    // Representa el cuadro de texto de la nota  
    const txtNota = document.querySelector("textarea");  
    let estado;  
    /* Esta función recoge un nuevo objeto estado y lo ajusta a los controles.  
     El objeto consta de dos propiedades:  
     - notas: Map que contiene las notas en forma de clave/valor  
     - seleccionada: Nota seleccionada dada por su clave  
     */  
    function setEstado(nuevoEstado) {  
      cbTitulos.textContent = "";  
      for (let nombre of Object.keys(nuevoEstado.notas)) {  
        const opcion = document.createElement("option");  
        opcion.textContent = nombre;  
        if (nuevoEstado.seleccionada == nombre) opcion.selected = true;  
        cbTitulos.appendChild(opcion);  
      }  
      txtNota.value = nuevoEstado.notas[nuevoEstado.seleccionada];  
      localStorage.setItem("Notas", JSON.stringify(nuevoEstado));  
      estado = nuevoEstado;  
    }  
    /* Se evalúa si hay algo en el almacenamiento local y en caso
```

```
contrario, se añade una primera nota de base*/
setEstado(JSON.parse(localStorage.getItem("Notas")) || {
  notas: { "Lista de la compra": "Zanahorias\nTomates" },
  seleccionada: "Lista de la compra",}
);
// Al cambiar el ítem del desplegable, se cargan los valores
cbTitulos.addEventListener("change", () => {
  setEstado({ notas: estado.notas, seleccionada: cbTitulos.value });
});
// Cuando se modifica el cuadro de texto de la nota, se va guardando el estado
txtNota.addEventListener("change", () => {
  setEstado({
    notas: Object.assign({}, estado.notas, {
      [estado.seleccionada]: txtNota.value,
    }), seleccionada: estado.seleccionada,
  });
});
// Al pulsar añadir, se inserta una nueva nota
document.querySelector("button").addEventListener("click", () => {
  let nombre = prompt("Nombre de nota");
  if (nombre)
    setEstado({
      notas: Object.assign({}, estado.notas, { [nombre]: "" }),
      seleccionada: nombre,
    });
});
</script>
</body>
</html>
```

El script obtiene su estado inicial del valor “Notas” almacenado en **localStorage** o, si falta, crea un estado de ejemplo que solo contiene una lista de la compra. La lectura de un campo que no existe en **localStorage** devuelve un nulo.

El método **setEstado** se asegura de que el DOM está mostrando un estado dado y almacena el nuevo en **localStorage**. Los manejadores de eventos van a llamar a esta función para pasar a un estado nuevo.

En el ejemplo se usa **Object.assign** que lo que hace es crear un objeto nuevo clonado del **estado.notas** antiguo pero con una propiedad añadida o sobrescrita. **Object.assign** toma su primer argumento y le añade todas las propiedades de otros argumentos, por lo que, si se le da un objeto vacío, rellenará un objeto nuevo.

## **4.4 DESARROLLO DE APLICACIONES MULTICLIENTE**

### **4.4.1 DOM EN APLICACIONES MULTICLIENTE**

Una aplicación multicliente es aquella que está diseñada para funcionar en distintos tipos de clientes, ya sean dispositivos o navegadores asegurando que la experiencia de usuario es coherente en todos ellos.

El DOM es vital en el desarrollo de este tipo de aplicaciones ya que permite la manipulación dinámica de la estructura y contenido de la interfaz de usuario en estos entornos.

Tipos de cliente no solo son los distintos navegadores clásicos para sistemas operativos de escritorio, también se pueden incluir los que trabajan sobre dispositivos móviles como Chrome para Android o Safari para iOS incluyendo aplicaciones híbridas (PWA – Progressive Web Apps). Por tanto, el DOM debe adaptarse al manejo de distintos tipos de pantalla, interacciones (táctiles, dispositivo apuntador o teclado).

Aquí entran conceptos como la reponsividad en la que el DOM se asocia a técnicas CSS como las Media Queries adaptando el contenido a diferentes tipos de pantalla y orientación.

Por ejemplo, el siguiente código detecta si la pantalla es táctil:

```
if ('ontouchstart' in window) {  
    // El dispositivo soporta pantallas táctiles  
    document.body.classList.add('dispositivo-tactil');  
} else {  
    // Dispositivo de escritorio  
    document.body.classList.add('dispositivo-escritorio');  
}
```

Anteriormente, se han mencionado las aplicaciones web progresivas (PWA) son aquellas diseñadas para ofrecer una experiencia similar a las nativas de escritorio, pero sobre tecnologías web estándar. En este tipo de aplicaciones, el DOM es fundamental para manejar la interfaz de usuario.

Una vez más, hay que mencionar a los frameworks y bibliotecas que ayudan en el desarrollo multicliente tales como React, Vue.js o Angular por su manejo eficiente del DOM.

### **4.4.2 EVENTOS DEL DOM EN DISTINTOS NAVEGADORES**

El comportamiento de los eventos del DOM puede variar en función del navegador utilizado, concretamente, teniendo en cuenta los distintos motores de renderizado existentes (Blink en Chrome, WebKit en Safari o Gecko en Firefox).

Como ya se ha citado, los navegadores antiguos tendían a ir más “a su bola” interpretando libremente cada evento del DOM. Las versiones modernas intentan ceñirse al W3C, con lo que esas diferencias cada vez son menores.

El W3C en la especificación del DOM de nivel 2, pone especial atención en los problemas del modelo tradicional de registro de eventos. En este caso ofrece una manera sencilla de registrar los eventos que se quieran sobre un objeto determinado.

Por ejemplo, el evento [addEventListener](#) está soportado por la mayor parte de los navegadores modernos teniendo un equivalente más antiguo utilizado por Internet Explorer llamado **attachEvent**.

Supóngase que se quiere controlar que un manejador de eventos funcione en navegadores antiguos. En el siguiente código se controla si existe `addEventListener` para lanzar un evento click o si, por el contrario, se trata de un navegador antiguo como IE y utiliza `attachEvent`:

```
// Evento click compatible con navegadores antiguos y modernos
const handleClick = (event) => alert("¡Has hecho clic!");

if (window.addEventListener) {
    // Navegadores modernos
    document
        .getElementById("miBoton")
        .addEventListener("click", handleClick);
} else if (window.attachEvent) {
    // Navegadores antiguos como IE
    document.getElementById("miBoton").attachEvent("onclick", handleClick);
}
```

Una buena práctica para evitar este problema es la utilización de bibliotecas como jQuery para normalizar dicha gestión de eventos.



## 4.5 INDEPENDENCIA DE LAS CAPAS DE IMPLEMENTACIÓN DE APLICACIONES WEB

Como ya se vio en la primera unidad, la arquitectura tipo de una aplicación web está estructurada en capas o niveles. Incluso aunque se opte por una opción muy básica, ya se distinguen dos capas: el frontal o **frontend** y el **backend**.

La idea de extender el número de niveles o capas y de que estas sean independientes ayuda en reducir acoplamiento (dependencia entre módulos o componentes) además de facilitar la escalabilidad y mantenimiento de una aplicación web. Otro modelo muy básico es el que distingue tres capas: presentación (frontal), lógica de negocio (reglas de validación, comportamiento, etc.) y capa de datos (acceso a bases de datos, ORMs, etc.).

El DOM pertenece a ese frontal o capa de presentación y, dado que su principal función es estructurar y organizar los elementos de la interfaz de usuario, no debe contener ningún tipo de lógica de negocio o de acceso a datos. De esta forma, tampoco la lógica de negocio debe estar atada a la posible visualización, por lo que no necesita conocer la estructura del DOM para poder ser reutilizada en otros posibles frontales (aplicación de escritorio, móvil, etc.).

¿Pero qué beneficios tiene este tipo de independencia entre niveles? A continuación, se enumeran las ventajas de una arquitectura en capas:

- Facilidad de mantenimiento: Mantener la independencia entre capas permite modificar o actualizar una parte de la aplicación (por ejemplo, el interfaz de usuario) sin que se vea afectado el resto de la lógica o el modelo de datos.
- Reutilización del código: La lógica de negocio puede reutilizarse en diferentes interfaces (web, móvil, aplicaciones de escritorio), ya que no está acoplada a la capa de presentación o al DOM.
- Mejor organización del código: Separar las responsabilidades entre las diferentes capas de la aplicación facilita una arquitectura más limpia y organizada.

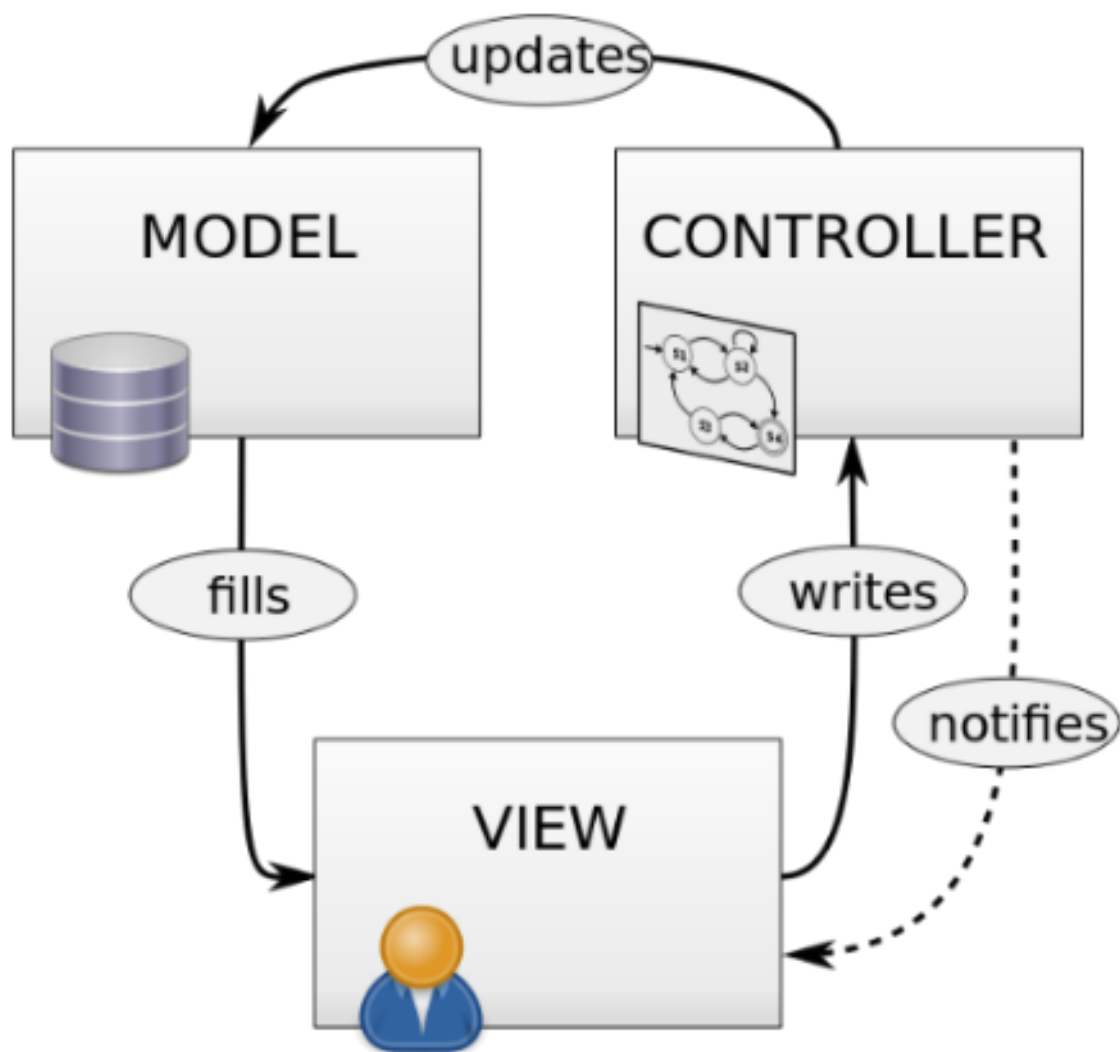
Otra arquitectura muy extendida es la MVC (Modelo – Vista – Controlador). En realidad, se puede considerar un gran patrón de diseño con el que se consiguen separar los datos (modelo) de la interfaz de usuario (vista), de manera que los cambios en esta última no afecten a los datos y viceversa, es decir, que estos puedan ser cambiados sin afectar a la interfaz de usuario.

Se divide en:

- Modelo: Es la parte del código responsable de contactar con los datos de la aplicación. Es un intermediario entre el origen de datos y la vista. Almacena y lee los datos además de asegurar su consistencia y validación. Contiene, por tanto, toda la lógica de negocio de la aplicación. Gracias a la separación de responsabilidades, el código llega a estar más estructurado además de ser más flexible a modificaciones.
- Vista: Es la parte de código responsable de la presentación al usuario de los datos obtenidos del modelo. La vista no contiene ninguna lógica de negocio, solo se encarga de

mostrar los datos, lo que facilita realizar cambios posteriores en la aplicación. Una vista puede cambiar el modelo solo si la modificación conlleva un cambio en la forma en la que se muestran los datos. De hecho, si se necesita realizar un cambio en el interfaz de usuario, solo impactará en la vista, pero no en la lógica de negocio.

- **Controlador:** Define la forma en que la interfaz reacciona a la entrada del usuario. Dependiendo de las acciones del usuario, actualiza el modelo y refresca la vista. También puede transferir el control a otro controlador. Además, permite el acceso a ciertas partes de la aplicación solo a usuarios autorizados.



## 2. MVC


Otras arquitecturas similares son **MVVM (Model-View-ViewModel)** en la que aparece un elemento llamado **ViewModel** que hace de puente entre vista y modelo y **MVP (Model View Presenter)** donde el nuevo modelo es el **Presenter** o Presentador el cual, tiene funciones similares a las del controlador y del **ViewModel**.

### 4.5.1 WEB COMPONENTS

Los **Web Components** o Componentes Web son un nuevo concepto que está encaminado a simplificar el codificado en desarrollo web. Este tipo de componentes permite crear nuevas etiquetas HTML personalizables y reusables que podrán funcionar en navegadores web modernos y se pueden usar con cualquier biblioteca JavaScript o Framework web que trabaje con HTML.

En esencia, los componentes web representan un esqueleto debajo de una misma estructura, pero cada elemento se puede construir de forma única a partir de él. Por ejemplo, en la figura siguiente pueden verse varios productos que pueden adquirirse en una conocida web de venta en línea:


**Resultados**  
Más información sobre estos resultados. Consulta la página del producto para ver otras opciones de compra. El precio y otros detalles pueden variar en función del tamaño y el color del producto.



Patrocinado ⓘ  
**Vistefly V15s MAX Aspiradora sin Cable, 580W 45Kpa Aspirador Escoba Potente 8 en 1 con Pantalla Táctil, Autonomía 70 Mins Modo Automático y Antienredos, para Suelo Alfombras Pelos Animales**  
★★★★★ 5.154  
600+ comprados el mes pasado  
**Black Friday**  
**169<sup>00</sup> €** Antes: 239,99€  
Pago a plazos disponible  
✓prime Envío 1 día  
Recogida GRATIS mañana, 25 de nov  
Añadir a la cesta

Clase de eficiencia energética: A+++

Opción Amazon



Patrocinado ⓘ  
**Rowenta Xpert 6.60 RH6838 - Aspiradora escoba versátil sin cable, con 3 funciones en 1 multiespacios, batería de litio, 2 velocidades, gatillo Boost, escoba ligera y portatil**  
★★★★★ 3.879  
3 mil+ comprados el mes pasado  
**Black Friday**  
**119<sup>99</sup> €** PVP: 199,99€  
o 30,00€ por pago en 4 cuotas con Amazon  
(sin intereses ni gastos financieros)  
✓prime  
Recogida GRATIS el mar, 26 de nov  
Añadir a la cesta

### 3. Componentes web en tienda en línea

Cada uno de los productos representados, sean o no distintos, comparten la forma de presentarse, es decir, todos llevan una foto, descripción, valoración, precio, etc.

Los componentes web generalmente se crean a partir de tres elementos independientes y mutuamente excluyentes que trabajan juntos:

- Elementos personalizados
- Shadow DOM. Se trata de una estructura que independiza los elementos creados en el componente con respecto a los de la página web que los contiene
- Plantillas HTML. Fragmentos de código que no se añadirán a la estructura HTML DOM sin que se solicite.

#### 4.5.1.1 Elementos personalizados

Hay una serie de elementos HTML que son recurrentes en cualquier diseño web, yendo desde `<div>` hasta `<section>` pasando por otros como `<time>` o `<aside>`. La idea es tener estos elementos agrupados y no tener que crearlos muchas veces. La creación de un elemento sigue cuatro pasos:

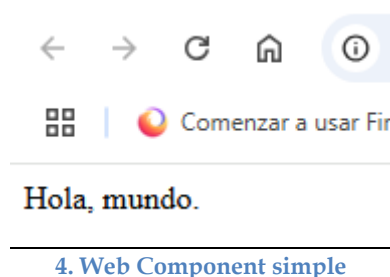
1. Creación de una clase que contendrá el elemento personalizado.
2. Agregar contenido a esa clase.
3. Hacer que el contenido de la clase sea exportable o simplemente disponible para cualquier página web.
4. Utilizar ese contenido y clase.

A continuación, se verá cómo crear un primer elemento personalizado. Para ello, lo primero que hay que hacer es crear un fichero JavaScript que en este caso contendrá el nombre del elemento, lo que no excluye tener varios componentes en un mismo fichero. El nuevo componente se llamará `MiTarjeta` e irá en el archivo `mi-tarjeta.js` cuyo código se muestra a continuación:

```
class MiTarjeta extends HTMLElement {  
  connectedCallback() {  
    this.innerHTML = "Hola, mundo.";  
  }  
}  
window.customElements.define('mi-tarjeta', MiTarjeta);
```

Como puede verse, se crea una clase que correspondería al nuevo elemento la cual hereda de **HTMLElement**. La nomenclatura CamelCasing es la que se utilizaría en estos casos para definir la clase, es decir, primera letra en mayúscula, eliminar espacios y cambiar a mayúscula la letra siguiente al espacio eliminado. Dentro de esa clase se crea un método llamado **connectedCallback** en el cual se inserta el contenido de la clase (paso 2). El tercer paso consiste en definir un `customElement` dentro de la ventana llamado `mi-tarjeta` que enlace directamente con la clase recién creada. Por último, solo queda incrustarlo en el código HTML, que es tan simple como insertar una etiqueta `<mi-tarjeta>`:

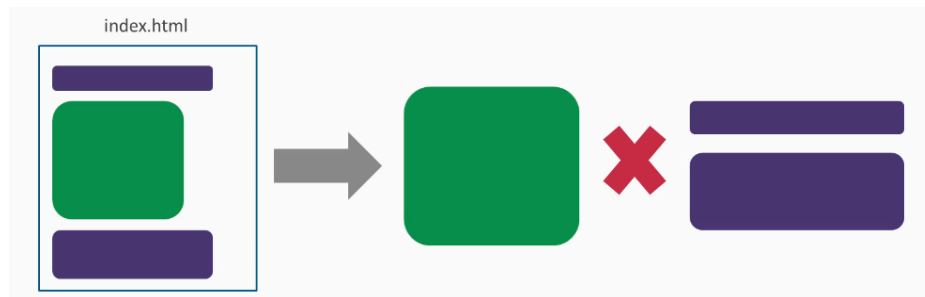
```
<body>  
  <script src="./mi-tarjeta.js"></script>  
  <mi-tarjeta></mi-tarjeta>  
</body>
```



#### 4.5.1.2 Shadow DOM

Otro de los elementos importantes en Web Components es el Shadow DOM, que no deja de ser una versión encapsulada del DOM. Los elementos dentro del componente y de este Shadow DOM no se verán afectados por el resto de los elementos DOM.

En la siguiente imagen, la zona en verde correspondería al Shadow DOM del componente mientras que la azul corresponde al DOM clásico.



5. Shadow DOM vs DOM

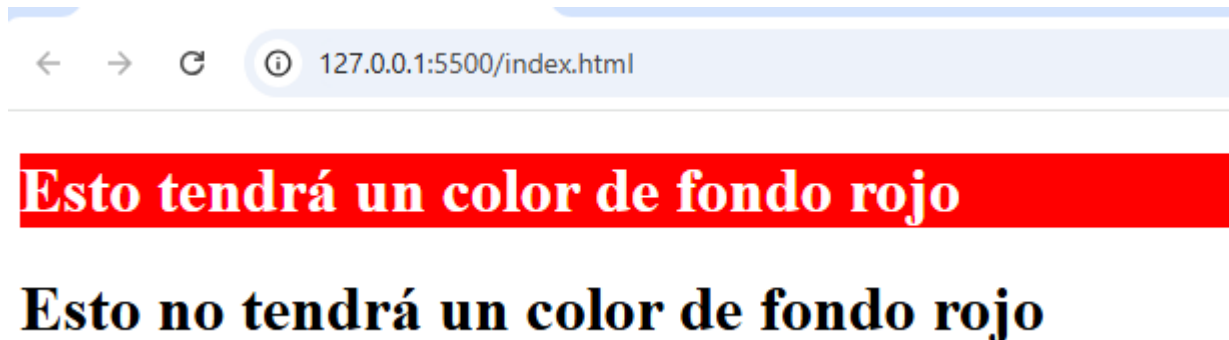
El uso de este Shadow DOM se ilustrará con un ejemplo a continuación. Los dos pasos necesarios para crear un Shadow DOM son:

1. Adjuntar un Shadow
2. Darle estilo y contenido

```
<html>
  <body>
    <div id="content"></div>
    <h1 id="header">Esto no tendrá un color de fondo rojo</h1>
    <script>
      const shadowRoot = document
        .getElementById("content")
        .attachShadow({ mode: "open" });
      shadowRoot.innerHTML = `
        <style>
          h1 {
            background: red;
            color: #fff;
          }
        </style>
        <h1 id='header'>Esto tendrá un color de fondo rojo</h1>`;
    </script>
  </body>
</html>
```

Este código presenta una serie de puntos muy interesantes. Consta de un div que alojará un contenido y será el que se sirva del Shadow DOM mientras que el encabezado H1 posterior de identificador header está relacionado con el DOM convencional. Lo primero que se hace es

adjuntarlo mediante **attachShadow** al div de identificador content. Se le da el modo **open** para poder acceder a dicho Shadow DOM mediante **shadowRoot**. Si se hubiera puesto **closed**, no podría accederse a él quedando la variable **shadowRoot** a **null**. Como puede verse en la figura, el elemento **content** tiene su propio DOM en el cual se le modifica el estilo mientras que el **h1** con el mismo **id** fuera del Shadow DOM no aplica dicho estilo.



6. Ejemplo de Shadow DOM

#### 4.5.1.3 Plantillas HTML

Las plantillas HTML son básicamente plantillas reutilizables de código que no van a ser cargadas y representadas hasta que no se solicite. El navegador va a ignorarlas hasta que se le pida que las tenga en cuenta. Por tanto, ejecutarlas en un navegador sin llamarlas no va a dar salida alguna. Hay que tener en cuenta que, aunque sean plantillas HTML, necesitan JavaScript para ser invocadas. Crear una plantilla HTML lleva tres pasos:

1. Crear un bloque de HTML y almacenarlo en un elemento **template**
2. Copiar la plantilla usando **importNode** y almacenarla en una variable
3. Añadir la variable a cualquier elemento en HTML

En el siguiente ejemplo se ilustra la creación de una plantilla. En primer lugar, dentro de la sección de script, se va a crear una lista con una colección de títulos de Sherlock Holmes con un pequeño subtítulo:

```
const titulos = [  
  {  
    titulo: "Las cinco semillas de naranja",  
    subtítulo: "La entrega de una carta inofensiva seguida de la muerte.",  
  },  
  {  
    titulo: "Estudio en escarlata",  
    subtítulo: "El Dr. Watson conoce a Sherlock Holmes",  
  },  
  {  
    titulo: "El perro de Baskerville",  
    subtítulo: "Un perro misterioso aterroriza a una ciudad.",  
  },  
];
```

Siguiendo las fases establecidas, el siguiente paso es crear la plantilla como tal. En este caso, se propone una muy sencilla en la que se mostrará el título seguido del subtítulo separados ambos por dos puntos. Esta plantilla puede ir en cualquier sección del código HTML, tanto en el head como en el body. Como puede verse, cada uno de los span está relacionado con una clase que es la que servirá posteriormente como selector.

```
<template id="plantilla-tarjeta">
  <p><span class="titulo"></span>: <span class="subtitulo"></span></p>
</template>
<div id="tarjeta-titulos"></div>
```

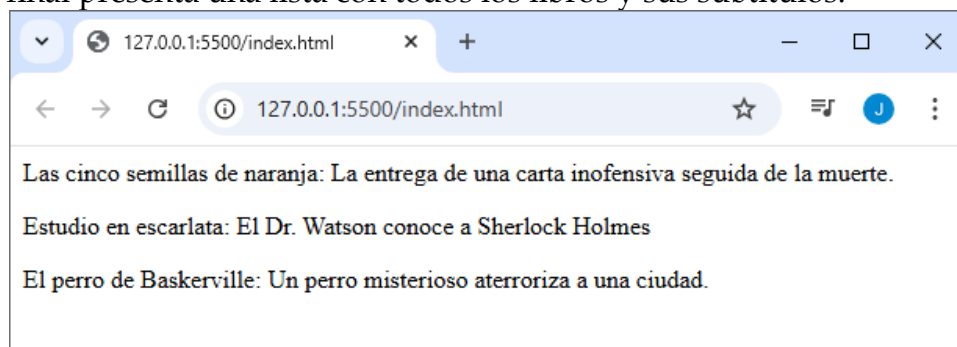
Además, se incluye un div donde realmente se materializará la plantilla llamado tarjeta-titulos. Volviendo al área JavaScript, en el siguiente código se puede ver el equivalente a los pasos dos y tres:

```
const fragmento = document.getElementById("plantilla-tarjeta");
titulos.forEach((t) => {
  const instancia = document.importNode(fragmento.content, true);
  instancia.querySelector(".titulo").innerHTML = t.titulo;
  instancia.querySelector(".subtitulo").innerHTML = t.subtitulo;
  document.getElementById("tarjeta-titulos").appendChild(instancia);
});
```

En este código “fragmento” representa el elemento plantilla. A continuación, se recorre la colección de títulos creada anteriormente y para cada uno de ellos se realiza lo siguiente:

1. Se declara la propia instancia de libro importando el contenido del fragmento. Esto se hace mediante **importNode**, método que permite crear una copia de un nodo desde un documento externo para ser insertado en el documento actual. Aquí, el documento externo es sencillamente el contenido de la plantilla representada como “fragmento”. El booleano indica si deben también importarse los descendientes del nodo.
2. Dado que la instancia no deja de ser una copia de la plantilla, se puede acceder a sus elementos mediante `querySelector` referenciando a las clases título y subtítulo.
3. Por último, se añade la instancia creada al div tarjeta-titulos. Es en ese momento donde se puede decir que se materializa el HTML del Web Component como instancias propias de la plantilla.

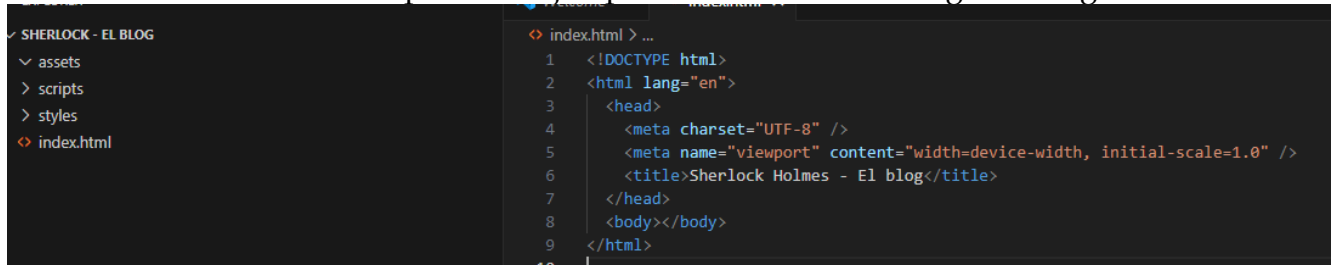
El resultado final presenta una lista con todos los libros y sus subtítulos:



## 7. Ejemplo de uso de plantilla HTML

Para ilustrar cómo reutilizar estas plantillas HTML, se verá un ejemplo más completo. En él se crearán dos tipos de componente: uno de tipo tarjeta o card y otro modal.

En primer lugar, lo más adecuado es crear una estructura de proyecto donde agrupar cada uno de los elementos necesarios para este ejemplo. Puede ser la de la siguiente figura:



8. Estructura de directorios para proyecto web components

Ahora, en head o body, se crea la plantilla para tarjetas:

```
<template id="plantilla-tarjeta">
  <style>
    .tarjeta {
      border: 1px solid #ddd;
      width: 200px;
      padding: 10px;
    }

    .encabezado {
      padding: 10px;
      border-bottom: 1px solid #ddd;
    }

    .contenido {
      margin-top: 10px;
    }
  </style>
  <div class="tarjeta">
    <div class="encabezado">
      <p class="titulo"></p>
      <span class="subtitulo"></span>
    </div>
    <div class="contenido">
      <span class="sinopsis"></span>
    </div>
  </div>
</template>
```

Como puede verse, además, se aplica un estilo a cada una de las secciones de esta tarjeta. Ahora, se crea un archivo JavaScript que va a recoger el código del Web Component donde se conjugan todos los aspectos vistos hasta el momento. Se alojará en scripts con el nombre de **tarjeta-blog.js**



```
const titulos = [
  {
    titulo: "Las cinco semillas de naranja",
    subtítulo: "La entrega de una carta inofensiva seguida de la muerte.",
  },
  {
    titulo: "Estudio en escarlata",
    subtítulo: "El Dr. Watson conoce a Sherlock Holmes",
  },
  {
    titulo: "El perro de Baskerville",
    subtítulo: "Un perro misterioso aterroriza a una ciudad.",
  },
];

class TarjetaBlog extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: "open" });
  }
  connectedCallback() {
    this.render();
  }

  render() {
    const { shadowRoot } = this;
    const nodoPlantilla = document.getElementById("plantilla-tarjeta");
    shadowRoot.innerHTML = "";
    if (nodoPlantilla) {
      titulos.forEach((t) => {
        const instancia = document.importNode(nodoPlantilla.content, true);
        instancia.querySelector(".titulo").innerHTML = t.titulo;
        instancia.querySelector(".subtitulo").innerHTML = t.subtitulo;
        shadowRoot.appendChild(instancia);
      });
    } else {
      shadowRoot.innerHTML =
        "<p>Fallo en la carga de Shadow Root. Por favor. inténtalo de nuevo más tarde.</p>";
    }
  }
}

customElements.define("tarjeta-blog", TarjetaBlog);
```

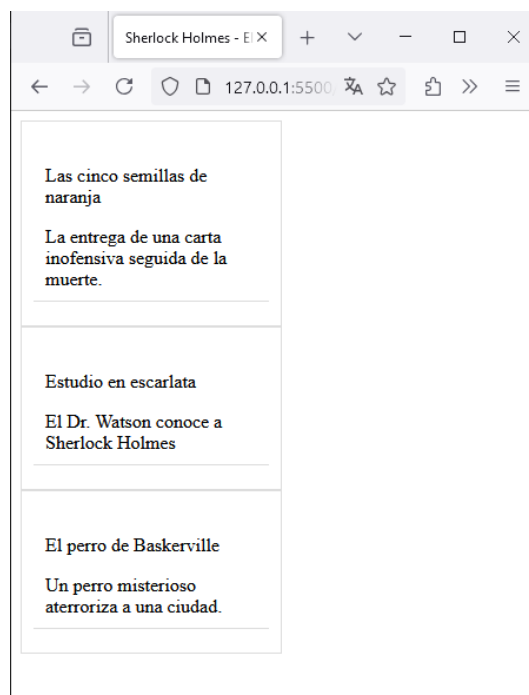
A continuación, se pasa a explicar el código adjunto en una serie de puntos:

1. Primero, se registra la colección de títulos con sus subtítulos. En otro contexto, esta información puede obtenerse de una API o de un script de un lenguaje de servidor (PHP, ASP.NET, etc.)
2. Se define el Web Component como TarjetaBlog. En el método **connectedCallback** se llama a otro método **render** el cual se encargará de representar el contenido del componente.
3. Dentro de render, lo primero que se hace es asociar la variable **shadowRoot** a la instancia de clase **this** mediante desestructuración. Por otra parte, se obtiene el nodo plantilla del elemento **template** que aparece en el documento HTML. Esta parte es muy interesante ya que se delega el aspecto y estructura del control a dicha plantilla.
4. Si se encuentra la plantilla, se realiza la operación vista anteriormente en la que se cubre cada elemento con el título y subtítulo de la colección. Cada uno de estos títulos se añade como hijo al Shadow DOM.
5. Para que el Shadow DOM se anexe al Web Component, en su constructor hay que hacer una llamada a **attachShadow** en modo abierto después de invocar al constructor de la clase base **HTMLElement**.
6. Se define el elemento tarjeta-blog con la clase recién creada.

Ahora solo queda insertar el elemento en el body de la página index además de añadir la referencia al script.

```
<script src="./scripts/tarjeta-blog.js"></script>  
<tarjeta-blog></tarjeta-blog>
```

El resultado final se puede ver en la figura:



9. Ejemplo completo de Web Component

## ÍNDICE DE FIGURAS

1. Árbol DOM.....	2
2. MVC .....	25
3. Componentes web en tienda en línea.....	26
4. Web Component simple .....	27
5. Shadow DOM vs DOM.....	28
6. Ejemplo de Shadow DOM.....	29
7. Ejemplo de uso de plantilla HTML.....	30
8. Estructura de directorios para proyecto web components .....	31
9. Ejemplo completo de Web Component .....	33

## BIBLIOGRAFÍA - WEBGRAFÍA

Ada Lovecode (2016). *Curso de Javascript - 4.03. Modelo de eventos del W3C*

<https://www.youtube.com/watch?v=nLyhVCCb9dc>

Shah, R. (2020) *Learn Practical Web Components Quickly*. <https://learning.oreilly.com/course/learn-practical-web/9781838649173/> Editorial O'Reilly.

Vigouroux, C. (2022) *Aprender a desarrollar con JavaScript*. Ediciones Eni. 4ª edición.

Haverbeke, M. (2023) *JavaScript elocuente*. Editorial Anaya. 3ª actualización.

MDN Web Docs. *Clases* (2024)

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