4. El DOM

4.1. Introducción y objetivos

El Document Object Model*,* más comúnmente conocido como DOM, es la representación que cada navegador web hace de los elementos de la página web que está mostrando al usuario. Se trata de una estructura en forma arbórea que, además, permite que se interactúe con la misma. Entre otras cosas, el DOM permite:

* Cambiar el contenido de los elementos.
* Mover elementos de lugar.
* Modificar sus atributos.
* Alterar significativamente su estilo.

Todo ello se hace posible gracias a JavaScript. Dicho de otra manera, JavaScript es el lenguaje de programación por defecto del DOM. En este caso, el DOM actúa como un *framework* (marco de trabajo) o también mal llamado librería (debería traducirse como biblioteca) con la que JavaScript adquiere cierta funcionalidad extra o novedosa.

El DOM es un *framework* concreto dispuesto para JavaScript por parte de los navegadores. Este *framework* permite a JavaScript interactuar con las páginas web.

Podemos resumir el concepto de *framework* como el conjunto de código informático y recursos asociados que posibilitan una cierta funcionalidad. Originalmente, JavaScript nació exclusivamente para operar sobre el *framework* DOM, el cual viene precargado en cada navegador web. Era, por tanto, un lenguaje de programación atado a una librería concreta, en el buen y mal sentido de la expresión (a veces resulta muy difícil desligar a un lenguaje de una librería concreta).

Estas librerías son una bendición y una condena para el lenguaje de turno. Desde cierto punto de vista, ayudan a elevarlo al dotarlo de funcionalidades, quizás asombrosas, pero desde otros ángulos, la comunidad de desarrolladores fija a fuego en su mente que tal lenguaje solo sirve para usar tal *framework*.

Esta bendición o condena dual le ha pasado a JavaScript en dos ocasiones en su historia. La primera fue, como ya hemos dicho, con el DOM. En la segunda, tuvo la culpa la famosa librería jQuery.

Afortunadamente para JavaScript, con la aparición de Node.js y la llegada de ES6, este lenguaje cobró relevancia propia, JavaScript se libró de las cadenas de la dependencia. A pesar de esta merecida libertad, es innegable que JavaScript tiene como principal cometido la manipulación del DOM y como principal escenario de uso el navegador web. Por eso debemos estudiar estos conceptos. En realidad, podríamos afirmar lo siguiente, el navegador web es un sistema operativo cuyo lenguaje de programación nativo es JavaScript.

Con más detalle, en este tema veremos:

* Qué es el DOM.
* Cómo localizar elementos.
* Cómo navegar por el DOM.
* Cómo extraer y cambiar las propiedades de los elementos.
* Cómo actualizar el DOM.

4.2. ¿Qué es el DOM?

Como decíamos, el Document Object Model o DOM representa a un documento HTML como un árbol de ítems o nodos interconectados.

Para el DOM, cualquier cosa en una página web es un nodo: desde las etiquetas HTML, junto con el texto dentro de las mismas, hasta incluso los atributos anexados a las etiquetas. La etiqueta (*tag*) HTML es el nodo raíz y cualquier otra sección del documento es un nodo «hijo» de este.

Por ejemplo, partamos desde el siguiente código HTML:



Figura1. Párrafo HTML. Fuente: elaboración propia.

Este tendría su representación como diagrama arbóreo DOM de la siguiente manera:

A diagram of a diagram

Description automatically generated

Figura 2. Diagrama arbóreo del DOM. Fuente: elaboración propia.

Como también hemos recalcado antes, el DOM no es JavaScript ni forma parte de él. De hecho, el DOM es accesible desde otros lenguajes de programación, pero qué duda cabe de que JavaScript es el más usado para esta tarea. En principio, cualquiera de los lenguajes que compilan (o, como decíamos en su momento, «transpilan») a JavaScript puede operar con el DOM.

El DOM nos provee de varios métodos para navegar por una página web. Te estarás preguntando: si un método es una función propia de un objeto, ¿de qué objeto estamos hablando cuando nos referimos a la palabra método?

Pues, efectivamente, el DOM como librería presenta a JavaScript un objeto concreto que a su vez define un espacio de nombres, tal objeto es document.

4.3. Un DOM virtual con JSDOM

¿Cómo es posible trabajar con un DOM y Node.js si este último es una implementación de JavaScript para el servidor? Node.js no tiene, de entrada, acceso a un DOM al que solicitarle cosas (¡no hay ningún navegador!). Pero antes de continuar conviene introducir otro concepto, el de API (*Application Programming Interface*).

Una API es el conjunto de funciones, variables, métodos, etc. que expone un *framework* o librería.

Es decir, un *framework* incorpora mucha más información de la que nos permite utilizar, de la que expone. Solo la API está a nuestra disposición. Un programa basado en Node.js no tiene acceso a la API del DOM. Sin embargo, esto podemos simularlo con el paquete JSDOM. Este *framework* para Node.js simula gran parte de la funcionalidad de un navegador con un DOM virtual.

Para instalar JSDOM tenemos que usar el gestor de paquetes NPM e instalar el paquete jsdom.

A black and white rectangle with a black border

Description automatically generated

Figura 3. Instalación de JSDOM. Fuente: elaboración propia.

Ahora ya podemos usar JSDOM con un código como este:

A computer code with text

Description automatically generated

Figura 4. Primeros pasos con JSDOM. Fuente: Elaboración propia

Si lo deseas, puedes ver el código fuente de JSDOM, ya que es público, en el recurso de A fondo, *Implementación de JSDOM.*

Lo que estamos consiguiendo es crear un DOM sin necesidad de un navegador y a partir de una cadena HTML sencilla ( <!DOCTYPE html><p>Hola!</p> ). Para mantener la coherencia con la aplicación del DOM en un navegador, hemos definido la variable intermedia  document , que coincide con el nombre oficial que tiene el objeto principal (y que define un espacio de nombres) del DOM cuando este es accedido desde el navegador.

Si estuviéramos en un navegador, tendríamos acceso directo a este objeto  document . Desde este espacio de nombres podemos acceder a todo el contenido HTML de una página web mediante la propiedad body. Ejemplo:



Figura 5. Obteniendo el body de un documento. Fuente: elaboración propia.

En este caso, se nos devuelve otro objeto ( cuerpoPagina ) de JavaScript. Este objeto, a su vez, posee otros métodos y propiedades. Por ejemplo, la propiedad  nodeType  nos dice qué tipo de nodo HTML es con un número entero:

A table with text and numbers

Description automatically generated

Tabla 1. Ejemplo de lo que devolvería la propiedad nodeType. Fuente: elaboración propia.

Otra propiedad es  nodeName , que devuelve el nombre del nodo:



Figura 6. Obteniendo el nombre de un nodo. Fuente: elaboración propia.

También hay otras propiedades sobre el objeto  document  que pueden resultar interesantes:

* document.images  devuelve una lista con todas las imágenes de una página web.
* document.links  devuelve una lista con todos los enlaces (elementos que tienen una propiedad  href cuyo contenido sea una URL).
* document.anchors  devuelve una lista con todos los elementos de etiqueta que a la vez tienen un atributo name.
* document.forms  devuelve un *array* con todos los formularios (elementos de etiqueta  form ) del documento web.

En realidad, estas listas o *array* de elementos (u otros nodos) de los que estamos hablando no son estrictamente *array*, sino una especie de *pseudoarrays* como los que se pueden ver al tratar un número variable de argumentos de una función. En esos casos, usamos la palabra especial arguments.

Al igual que con arguments, las listas de nodos permiten usar la notación de índice, por ejemplo, unArrayDeNodos[1].

También tienen una propiedad  .length  que devuelve el número de elementos. Sin embargo, y nuevamente, al igual que arguments, los *arrays* de nodos no tienen otros métodos propios de los *arrays*, tales como slice, join, etc. Afortunadamente, desde ES6, es posible pasar de una lista de nodos a un array normal y corriente gracias al método  Array.from(listaNodos) . Ejemplo:



Figura 7. Creación de *array* a partir de una lista. Fuente: elaboración propia.

O podemos usar el operador de expansión (...) que hemos visto anteriormente:



Figura 8. Usando el operador de expansión para construir un *array*. Fuente: elaboración propia.

Localizar elementos mediante su identificador

El método  getElementById()  del objeto document hace exactamente lo que dice: devuelve un elemento si le pasamos su identificador o id. Todo elemento (nodo) del DOM puede tener una marca identificadora única que se etiqueta mediante el atributo id. Ejemplo:



Figura 9. Uso de getElementById. Fuente: elaboración propia.

Localizar elementos mediante su etiqueta o tag

El método  getElementsByTagName()  devuelve una lista de nodos con todos aquellos elementos de una etiqueta determinada (h1, h3, li, ul, img, p, div, etc.). Fíjate que la palabra «Elements» de getElementsByTagName  está ahora en plural (en contraste con su versión en singular de getElementById ).

Esto es así porque, por descontado, una página web puede tener uno o varios elementos con la misma etiqueta, como imágenes (img), párrafos (p) o listas sin orden (ul). Veamos un ejemplo:



Figura 10. Uso de getElementsByTagName. Fuente: elaboración propia.

Ahora podemos navegar por el array de elementos sin mayor problema como si de un *array* real se tratara (que ya hemos visto que se trata más bien de un *pseudoarray*):



Figura 11. Obtención del primer elemento del *pseudoarray*. Fuente: elaboración propia.

Localizar elementos por su clase

Con el método  getElementsByClassName()  podemos obtener una colección de elementos que comparten la misma clase. La clase en HTML es un atributo que agrupa ciertos elementos con objetivos parecidos.

A computer code with text

Description automatically generated

Figura 12. Obteniendo elementos a partir de su clase. Fuente: elaboración propia.

### **4.4. Query selectors**

Gracias a los métodos  querySelector(selector)  y  querySelectorAll(selector)  del objeto document  es posible realizar consultas al DOM utilizando selectores idénticos a los selectores de CSS. Otros selectores que se pueden utilizar son los siguientes:

A blue and white text box

Description automatically generated with medium confidence

Tabla 2. Otros selectores. Fuente: elaboración propia.

El primero de los métodos, . querySelector(selector) , solo devuelve el primer elemento encontrado. Es, por tanto, perfecto para buscar elementos por su identificador único.

Ejemplo:

A white rectangular sign with black text

Description automatically generated

Figura 13. Query selector #. Fuente: elaboración propia.

El método  .querySelectorAll(selector)  devuelve todos los elementos que cumplen un selector, a partir del elemento raíz del DOM (el objeto document), por ejemplo:



Figura14. Query selector All. Fuente: elaboración propia.

Esto devuelve todos los elementos (una lista de nodos) cuya clase es saludo. En realidad, el método querySelector()  puede invocarse sobre cualquier nodo, no solo  document . Por ejemplo, supongamos que seleccionamos primero una lista ordenada a partir de su etiqueta y su id:



Figura 15. Query selector sobre un nodo distinto de document 1 . Fuente: elaboración propia.

Pues ahora podemos buscar otros nodos dentro del nodo  listaAlumnos  de la siguiente manera:



Figura 16. Query selector sobre un nodo distinto de document 2 . Fuente: elaboración propia.

### **4.5. Navegación y modificación de atributos**

Además de los métodos vistos anteriormente que nos permiten realizar consultas al DOM para obtener uno o varios elementos de este, existen **otras propiedades de los nodos del DOM** que nos facilitan la navegación por el árbol.

Algunas de estas propiedades son:

A screenshot of a phone

Description automatically generated

Tabla 3. Propiedades de navegación. Fuente: elaboración propia.

Un ejemplo sería:

A white rectangular sign with black text

Description automatically generated with medium confidence

Figura 18. Uso de firstChild. Fuente: elaboración propia.

También es posible leer y modificar el valor de los atributos de los nodos del DOM mediante los métodos .setAttribute(atributo, valor)  y  .getAttribute(atributo) . Estos nos permiten establecer o leer el valor de un atributo. Por ejemplo, con el siguiente código estaríamos modificando la clase de un nodo y luego la estaríamos leyendo nuevamente:

A close-up of a sign

Description automatically generated

Figura 19. Uso de setAttribute y getAttribute. Fuente: elaboración propia.

### **4.6. Creación y modificación de nodos**

El framework del DOM incluye métodos para modificarlo **en tiempo real.** Estos cambios se aplican en la página web mientras es visualizada y el usuario interactúa con la misma. Es responsabilidad del navegador el estar atento a estos cambios y aplicarlos. Veamos cómo se consigue este dinamismo.

Para empezar, tenemos la instrucción  .createElement()  que acepta como parámetro una etiqueta y devuelve un elemento de dicho tipo (correspondiente a dicha etiqueta). Por ejemplo:



Figura 20. Método createElement. Fuente: elaboración propia.

Hasta este punto el elemento ( elementoHola ) está vacío. Para completarlo con contenido (por ejemplo, texto) tenemos que crear un nodo de texto. Para crear este tipo de nodos, usamos el método .createTextNode(texto) . Por ejemplo:



Figura 21. Método createTextNode. Fuente: elaboración propia.

Hasta ahora tenemos un nodo de tipo  h1 (hola)  y un nodo de texto  (textoHola) , pero no están conectados. Es necesario **enlazarlos.** Esto lo conseguimos con el método  .appendChild(nodoTexto) llamado sobre el elemento elementoHola:



Figura 22. Método appendChild. Fuente: elaboración propia.

El proceso podría resumirse así:

* Crear un nodo elemento (o raíz) con el método  createElement  (etiqueta).
* Crear un nodo de texto (en este caso) con el método  createTextNode  (texto).
* Enlazar ambos nodos mediante el método  nodoRaiz.appendChild  (nodoTexto).

Sin embargo, con esto no es suficiente. Disponemos de un nodo completo, pero no está situado en **ningún punto del árbol de nodos actual.** Está «suelto» al no tener ningún nodo padre.

Como habrás podido adivinar, esto se consigue también con el método con .appendChild(nodoElemento) , pero ahora invocado, en lugar de sobre el objeto document, sobre otro elemento del DOM cualquiera  (otroElemento)  en el que queramos que nodoElemento aparezca incrustado como hijo (child).

Además de  otroElemento.appendChild(nodoElemento) , tenemos  insertBefore()  e

insertAfter() .

Para borrar elementos tenemos  unElemento.removeChild(nodoElemento) , donde

nodoElemento  es un nodo que cuelga de  unElemento .



Figura 23. Propiedad innerHtml. Fuente: elaboración propia.

Por último, puedes encontrar a continuación el enlace al vídeo Interactuando con el DOM en el que se repasarán algunos de los conceptos que se han expuesto a lo largo del tema.