Js en la web- DOM y comportamiento del formulario

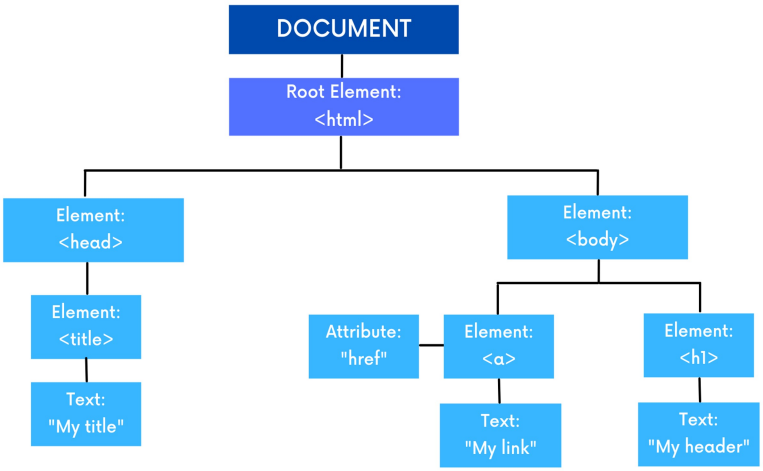
jueves, 1 de diciembre de 2022 21:06

DOM

La traducción del acrónimo inglés DOM (Document Object Model) significa Modelo de Documento de Objeto; se trata de modelar todo el HTML.

Estructura

El DOM es como un árbol genealógico, pero en forma invertida. El elemento que precede al document es el window, que no es más que la ventana del navegador. En su estructura, document se encuentra en la parte superior como un objeto global y su elemento raíz es la etiqueta html y todos los demás descienden de él a través de sus ramas (branches).



Fuente: Representación adaptada de W3Schools

La etiqueta html, objeto padre, tiene dos objetos hijos: el head y el body (la cabeza y el cuerpo). Los objetos que siguen a las ramas inferiores se denominan child, y los de arriba, parent. La etiqueta head es parent de la etiqueta title, y el body es parent de las etiquetas a y h1, y así sucesivamente, según la jerarquía. A partir de las etiquetas, derivan los atributos, y de estos, sus valores.

Dónde se inserta?

Y la pregunta que surge es: ¿pero el DOM es parte de HTML o de JavaScript? De hecho, de ninguno — él es generado por el browser. Al cargar la página, el navegador crea el documento, la interfaz, y Javascript usa el DOM para conectarse al HTML.

Para comunicarse entre ellos, es necesario insertar la etiqueta script en el archivo HTML, y como buena práctica, debe ser antes del cierre de la etiqueta body para que los scripts se carguen después del código base.

Puedes hacerlo de dos maneras: escribiendo el código JavaScript dentro de la propia etiqueta script, o insertando la ruta relativa del archivo externo. También como buena práctica, la segunda opción es la más recomendada para la separación de responsabilidades y un mejor mantenimiento del código.

```
<script>
  alert("¡Hola, Mundo!")
</script>
<scriptsrc="script.js"></script>
```

Hay varias formas de navegar dentro del DOM, en JavaScript usamos el objeto document y a través del punto accedemos a las propiedades y métodos, siendo posible seleccionar, cambiar, borrar y crear elementos a los componentes del sitio web, según la estandarización creada por W3Schools. Para realizar estas acciones disponemos de algunos métodos, tales como:

- document.getElementById();
- document.getElementsByClass();
- document.getElementsByTagName();

- document.querySelector();
- document.querySelectorAll();
- document.createElement();
- element.addEventListener();

Con document.querySelector, por ejemplo, entre las opciones que ofrece, podemos cambiar el texto en el documento HTML

```
<body><h1>¡Hola! </h1>
<scriptsrc="script.js"></script></body>
```

```
document.querySelector("h1").innerText = "¡Hola, Mundo!"
                        o
document.querySelector("h1").textContent = "¡Hola, Mundo!"
```

COMO FUNCIONAN LOS FORMULARIOS

- Utilizamos **data-attributes** en vez de las clases por si son modificados en un futuro ej:

```
<button type="submit" class="btnCreate" data-form-btn>
     Agregar <i class="fas fa-plus-circle"></i>
</button>
```

Y para llamarlos en JS utilizamos:

```
const btn = document.querySelector("[data-form-btn]");
```

Para utilizar los eventos neceitamos escucharlos con **Listenners**
¿Cuáles son las tres cosas que necesitamos para utilizar un escuchador de eventos?
Necesitamos el tipo del evento, el elemento que recibirá este evento y la acción que pasará cuando el evento sea disparado

- **Todo los elementos son objetos por lo tanto tienen metodos/Funciones**

Por ej: btn que es el tipo de evento; addEventListener el metodo el cual es una funcion que tiene 2 parametros; el primero es el evento que esta entre ""; y le sigue una duncion que va a tener la accion que queremos que haga.

```
btn.addEventListener("click", function(){
     Contenido de la funcion
})
```

Capturando el valor de input

```
let input = document.querySelector("[data-form-input]");

btn.addEventListener("click", function(){
    console.log(input.value);
})
```

Comportamiento del formulario

- **Como previnir el funcionamiento que hace que se recargue la pagina cada vez que hacemos click en agregar**

```
btn.addEventListener("click", function(evento){
    evento.preventDefault();
    console.log(input.value);
})
```

Existen las arrow function o funciones anonimas:

Usar arrow functions es la preferencia actual al escribir Javascript moderno, estas te permiten escribir métodos más concisos o simplificaciones de una línea más legibles aprovechando las características de retorno implícito y el no uso de paréntesis.
Al principio cuesta visualmente acustumbrarse ya que se usan sobre todo en una única sentencia, es decir, todo el cuerpo de la función es una única línea

```
btn.addEventListener("click", (evento) => {
    evento.preventDefault();
    console.log(input.value);
})
```

Y para que quede mas limpio el codigo que es muy importante
Guardamos la funcion en una constante:

```
 const createTask = (evento) => {
    evento.preventDefault();
    console.log(input.value);
};

btn.addEventListener("click", createTask)
```

Agregando tarea

- **Como limpiar el input una vez presionado el btn**

```
const createTask = (evento) => {
    evento.preventDefault();
    const input = document.querySelector("[data-form-input]");
    const value = input.value;
    input.value = "";
    console.log(value);
};
```

Esto guarda el valor del input en una variable "value" y luego cambiamos el valor del input a vacio.

- **Agregando la tarea del input al principio.**

```
 //task es ele elemento li donde va nuestro texto o tarea
const task = document.querySelector("[data-task]");
//backlist (es una template string que utiliza comillas invertidas ``)
//Con esto nosotros podemos combinar etiquetas HTML con variables de JS
//Y para usar una variable de js en los template string usamos ${variable}
const content = `
    <div>
    <i class="far fa-check-square icon"></i>
    <span class="task">${value}</span>
    </div>
    <i class="fas fa-trash-alt trashIcon icon"></i>
    </li>`
//Y ahora agregamos el content osea el codigo HTML al elemento li que es el task
//Con la propiedad innerHTML le podemos asignar codigo html a task
task.innerHTML = content;
```

Este codigo solo hace que la tarea que escribamos remplaze a la que esta primera, y nosotros tenemos que hacer que cada vez que escribamos una tarea sea añadida con las demas.

- **Añadiendo las tareas(crear una nueva card por cada elemento que va poniendo el usuario)**

Primero limpiamos el codigo de nuestro ul, osea borrar el li, ya que es codigo estatico.
Creamos una nueva const de nuestro elemnto padre osea el ul

```
const list = document.querySelector("[data-list]")
```

Como ya no va funcionar nuestro query selector del data task. Tendremos que crear un li dentro de nuestro ul, con la propiedad:

```
const task = document.createElement("li")
```

Luego a ese task hay que agregarle una clase para poder usar css

```
 task.classList.add("card");
```

En este punto ya tenemos el el content que se agrega al elemnto task que es nuestras li, pero nos falta agregar esa li a nuestro elemento padre que es el ul. Y eso lo hacemos con la prodiedad:

```
list.appendChild(task)
```

Y con este codigo podemos ver que cada vez que agregamos una tarea se va creando nuevo codigo html dinamicamente.

```
const createTask = (evento) => {
    evento.preventDefault();
    const input = document.querySelector("[data-form-input]");
    const value = input.value;
    input.value = "";
    const list = document.querySelector("[data-list]")
    const task = document.createElement("li")
    //A este li tenemos que agregarle una clase ya que tiene codigo css tambien
    task.classList.add("card");
    const content = `
        <div>
        <i class="far fa-check-square icon"></i>
        <span class="task">${value}</span>
        </div>
        <i class="fas fa-trash-alt trashIcon icon"></i>
        </li>`
    task.innerHTML = content;
    //Con appendChild estamos diciendo al elemnto seleccionado agregarle un hijo
    list.appendChild(task)
    console.log(content);
};
```

Todos los elementos en nuestro árbol de DOM son nodos y todos los nodos pueden ser accedidos vía JavaScript. Los nodos pueden ser eliminados, creados o modificados. Durante el curso utilizamos el método appendChild que siempre es implementado al final del nodo, para colocar un nodo hijo dentro del nodo padre.

Existen otros métodos que podemos utilizar para manipular nodos:

- **insertBefore**(padre, hijo): Coloca un nodo antes del otro
- **replaceChild**(elemento1, elemento2): Sustituye el nodo del elemento 1 por el nodo del elemento 2
- **removeChild**(elemento): Remueve un nodo del árbol

- **Creando el boton de concluido y borrado**

Lo que vamos a hacer ahora es crear los conos de borrado y conluido en JS
Necesitamos crear una funcion que nos retorne este elemento`<i class="far fa-check-square icon"></i>.` y podamaos modificarlo.

```
const checkComplete = () => {
    const i = document.createElement("i");
    i.classList.add("far fa-check-square icon")
    return i
}
```

La funcion checkComplete nos devuelve codigo html por lo tanto no lo podemos insertar en el content ya que es un template string, necesitamos que sea string, asi que hay que crear cada elemeneto del content para que funcione nuestra funcion.
Esto lo hacemos primero creando el elemento padre del content en este caso el div:

```
const taskContent = document.createElement("div")
```

Luego le agregamos el elemento hijo que le sigue en este caso sera nuestra funcion que retorna el check o i:

```
taskContent.appendChild(checkComplete)

Ahora a nuestro task le agregamos como elemento hijo el task content
task.appendChild(taskContent)

Una vez hecho eso nos faltaria el titulo de la tarea y e icono del la basura
Para el titulo hacemos los mismo creamos el elemento en una variable, le ponemos la clase, e
insertamos la variable con el texto.
const titleTask = document.createElement("span")
titleTask.classList.add("task")
titleTask.innerText = value
```

Una vez hecho todo eso hay que agregarselo al elemento padre en este caso el div osea el taskContent
```
taskContent.appendChild(titleTask)
```

Lo mismo hacmeos con el boton de borrado

```
const deleteIcon = () => {
    const i = document.createElement("i");
    i.classList.add("fas", "fa-trash-alt", "icon")

    return i
}
```

- **Concluir tarea**

Ahora creamos las funciones para los botones para que hagan lo que tiene que hacer, en este caso completar e task y borrar el task
Y las agregamos a las funciones donde creamos los botones

```
const completeTask = (event) => {
    const element = event.target
}
const deleteTask = (event) => {
    const element = event.target
}
```

El **event.target** es para diferenciar cada objeto y que al precionar un elemento solo cambie el precionado
Y ahora para que el icono aparezca completado hay que cambirle la clase porque en esete caso la clase es un **fontawesone** y le da el stilo al icono, entonce lo cambiamos de **far a fas** para que aprezca completado al hacerle click. Y le agregamos la clase que esta en css para que aprezca en azul el icono, que ya esta completado.

```
const completeTask = (event) => {
    const element = event.target
    element.classList.add("fas");
    element.classList.add("completeIcon")
    element.classList.remove("far");
}
```

Y tambien tendriamaos que al aparetar el boton marcar como com`letado y no completado y para hacer eso, como existe add y remove tambien existe una funcion comidin es **toggle** que lo que haces es que **si encuentra la clase la borra y si no esta la agrega.** Entonce nuestro codigo nos quedarias asi:

```
const completeTask = (event) => {
    const element = event.target
    element.classList.toggle("fas");
    element.classList.toggle("completeIcon")
    element.classList.toggle("far");
}
```

Para eliminar la tarea debemos usar una funcion que llame al evento y que nos traiga el li osea la card, para eso podemos usar path que es un array donde tiene todos los padres e hijos hasta el event. Luego con la funcion **remove** eliminamos esa card

```
    const deleteTask = (event) => {
        const card = event.path[1];
        card.remove()
    }
```

Y luego lo agregamos a la funcion donde creamos el icono:

```
    const deleteIcon = () => {
        const i = document.createElement("i");
        i.classList.add("fas", "fa-trash-alt", "icon")
        i.addEventListener("click", deleteTask);
        return i
```

```
}
```

- **Javascrip IIFE**
- Una IIFE (Immediately Invoked Function Expression) constituye un patrón de diseño usado comúnmente en Javascript (por bibliotecas, como jQuery, Backbone. js, Modernizr, y muchas más) **para encapsular un bloque de código dentro de un ámbito local.**
  Son funciones que se ejecutan tan pronto como se definen. Es un patrón de diseño también conocido cómo función autoejecutable

  El usario tiene acceso a las funciones que definimos en JS. Lo que nosotros queremos es que no esten a la alcanze del usario
  Y para definir esta funcion =

```
( () => {

    Todo el codigo JS

}) ()

Los ultimo parentesis son para llamar a la funcion inmediatamnete
```

- **Ultimos detalles**

  Implementar modulos:
  Lo que vamos hacer es crear modulos los que se van a encargar de contener cierta parte de codigo
  Por ej: Creando una carpeta que se llame components y dentro de esa carpeta crear archivos js
  Como checkComplete.js y deleteIcon.js.
  Una vez separado el codigo hay que importarlo en scrip.js.
  Una manera de hacerlo es en los archivos de js creado lo exportamos y dentro del archivo original script.js importamos De esta manera:

Archivos	Codigo
CheckComplete.js	`export default checkComplete`
DeleteIcon.js	`export default deleteIcon`
Script.js	`import checkComplete from "./components/checkComplete";` `import deleteIcon from "./components/deleteIcon";`

  Y dentro de HTML donde esta el script tenemos que definir que el script es tipo modulo
  `<script type="module" src="script.js"></script>`