JAVASCRIPT

CLASE 7

Material complementario

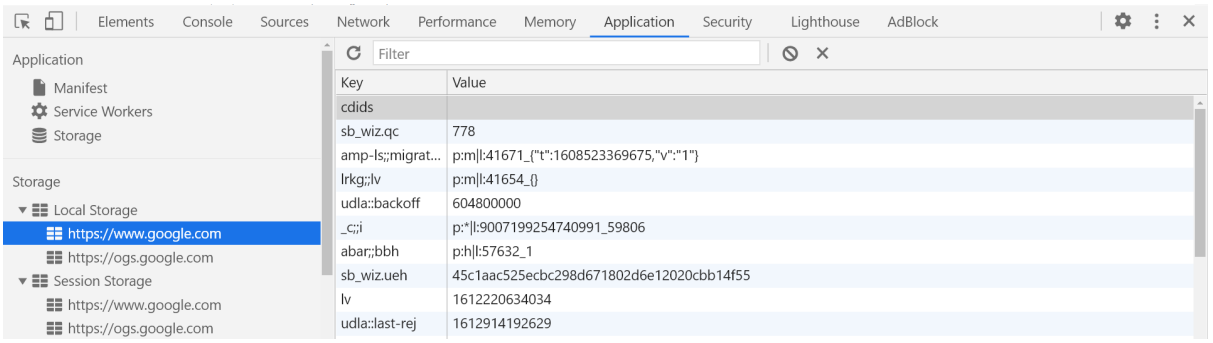
STORAGE Y JSON

CODER HOUSE

JavaScript: storage

Storage: definición

El storage es un medio de almacenamiento en el cliente, lo cual implica que podemos utilizarlo para guardar información de la apliaciòn en el navegador del usuario. Es común que aplicaciones web empleen el storage para conservar datos de usuario mientras utilizan el sitio. Podemos comprobar nosotros/as mismos/as que Google emplea dicho recurso desde la consola que desarrolló, en la pestaña ***aplicación***, sección ***Storage:***



La información almacenada en el storage tiene la estructura clave-valor:

- Clave: identificador alfanumérico que permite obtener un valor almacenado en el storage.
- Valor: un string con cierto formato (DOMString) que representa la informaciòn almacenada.

Podemos pensar al storage como una tabla, donde cada fila está compuesta por dos celdas: la primera es el término para identificar la información guardada, y la segunda la información en sí.

Tenemos dos tipos de storage: uno identificado como temporal, cuya información almacenada existe si la pestaña actual en uso se encuentra abierta, conocido como

SessionStorage; y otro donde la información almacenada se conserva, incluso si se cierra el navegador, conocida como LocalStorage.

LocalStorage

Los datos almacenados en localStorage (variable global preexistente) se almacenan en el navegador de forma indefinida, o hasta que se borren los datos de navegación del browser. La información persiste aunque reiniciemos el navegador, e incluso el sistema operativo. Además, los datos se comparten entre pestañas, permitiendo que los valores almacenados puedan ser utilizados y actualizados en distintas pestañas del navegador.

Para almacenar un valor en localStorage, y teniendo en cuenta que es un objeto, podemos emplear la llamada al método ***setItem*** de la siguiente manera:

```
// Método ->  localStorage.setItem(clave, valor)
// clave = nombre para identificar el elemento
// valor = valor/contenido del elemento
localStorage.setItem('bienvenida', '¡Hola Code!');
localStorage.setItem('esValido', true);
localStorage.setItem('unNumero', 20);
```

Para recuperar la información almacenada en el localStorage, es necesario emplear el nombre de clave como parámetro del método ***getItem***, lo cual no retornara el valor asociado:

```
let mensaje =  localStorage.getItem('bienvenida');
let bandera =  localStorage.getItem('esValido');
let numero  =  localStorage.getItem('unNumero');

console.log(typeof mensaje); //string;
console.log(typeof bandera); //string;
console.log(typeof numero);  //string;
```

Hay que tener presente que toda la información recuperada del storage es un string, y en consecuencia es necesario parsear algunos tipos de datos para emplearlos adecuadamente.

SessionStorage

La información almacenada en sessionStorage (variable global preexistente) permanece en el navegador hasta que el usuario cierra la ventana. Los datos sólo existen dentro de la sesión actual. lo cual implica que la información en el sessionstorage no se comparte entre pestañas.

Si abrimos otra pestaña con la misma página, esta tendrá un sessionStorage exclusivo. No obstante, es posible acceder esta información desde elementos embebidos, como pueden ser iframes en la pestaña (asumiendo que estos tengan el mismo origen).

Para almacenar información en el SessionStorage, empleamos la llamada al método setItem de la siguiente manera:

```
// Método ->  sessionStorage.setItem(clave, valor)
// clave = nombre del elemento
// valor = Contenido del elemento
sessionStorage.setItem('seleccionados', [1,2,3]);
sessionStorage.setItem('esValido', false);
sessionStorage.setItem('email', 'info@email.com');
```

Para recuperar la información almacenada en el sessionStorage, también es necesario emplear el nombre de clave como parámetro del método getItem. Notemos que en este ejemplo ,el campo de clave 'seleccionados' es un array al momento de almacenarse, pero se recupera del storage como string; en consecuencia, es necesario emplear el método split para recuperar la información, asociando el tipo de dato correcto.

```
let lista    =  sessionStorage.getItem('seleccionados').split(",");
let bandera = (sessionStorage.getItem('esValido') == 'true');
let email    =  sessionStorage.getItem('email');

console.log(typeof lista);    //object ["1","2","3"];
console.log(typeof bandera); //boolean;
console.log(typeof email);    //string;
```

Lo mismo ocurre con el dato con clave ***esValido***, con el cual para recuperarlo como valor booleano desde el storage (como una string 'true' o 'false') empleamos una comparación que nos permite obtener el tipo de dato correcto.

Acceso de tipo objeto y recorrido

Si tenemos presente que localStorage y sessionStorage son objetos globales, es posible crear y acceder a las claves como si fueran propiedades de un objeto. Pero esto no es recomendable por dos motivos:

1. Hay eventos asociados a la modificación del storage al emplear setItem, que no funcionan con la asignación de tipo objeto.
2. No es posible emplear algunas palabras como claves (por ejemplo length o toString) cuando se crean mediante el acceso de tipo objeto, pero con los métodos getItem y setItem estos términos funcionan correctamente:

```
//Guarda una clave
localStorage.numeroPrueba = 5;

//Leer una clave
alert( localStorage.numeroPrueba ); // 5

let clave = 'toString';    //toString método reservado
localStorage[clave] = "6"; //No se guarda este dato
```

Dado que local y session storage son objetos, también es común pensar que podemos recorrer las claves creadas por nosotros con ***for...in***. Pero esta forma de acceso no es muy eficaz, dado que se listan todas las propiedades existentes en el objeto, incluso las existentes por defecto.

Para distinguir correctamente las claves creadas en el storage por el/la programador/a, debemos emplear el método key, por ejemplo la siguiente manera:

```
//Ciclo para recorrer las claves almacenadas en el objeto localStorage
for (let i = 0; i < localStorage.length; i++) {
    let clave = localStorage.key(i);
    console.log("Clave: "+ clave);
    console.log("Valor: "+ localStorage.getItem(clave));
}
```

Otra opción de recorrido de claves es emplear un ***for...of*** sobre las propiedades de los objetos storage usando keys(), luego distinguimos cada propiedad iterada como clave del storage usando el método key().

Eliminar datos del storage

Para eliminar la información almacenada en sessionStorage o localStorage tenemos dos opciones:

- Eliminación por clave: se elimina un único valor en el storage detallando la clave como parámetro del método removeItem.
- Vaciado: eliminamos toda la información existente en el storage con el método clear().

Veamos un ejemplo de cada implementación:

```
localStorage.setItem('bienvenida', '¡Hola Code!');
sessionStorage.setItem('esValido', true);

localStorage.removeItem('bienvenida');
sessionStorage.removeItem('esValido');
localStorage.clear()    //elimina toda la información
sessionStorage.clear() //elimina toda la información
```

JSON

Mencionamos anteriormente que tanto la clave como el valor se almacenan como string. Esto implica que cualquier otro tipo de dato, como un número o un objeto, se convierte a cadena de texto automáticamente al momento de guardarse con setItem. Esta situación puede generar ciertas inconsistencias en los datos almacenados, principalmente en los objetos, ya que su transformación a string por defecto es la cadena "[object Object]", como vemos en siguiente ejemplo:

```
const producto1 = { id: 2, producto: "Arroz" };
localStorage.setItem("producto1", producto1); // Se guarda
[object Object]
```

Entonces necesitamos un formato que nos permita almacenar la información de un objeto en una cadena de texto: ese es JavaScript Object Notation (JSON). Se trata de un formato de texto plano, que sirve para representar datos estructurados con la sintaxis de objetos de JavaScript. Es muy utilizado para enviar y almacenar datos en aplicaciones web.

Aunque es muy parecido (casi similar) a la sintaxis de JavaScript, puede ser utilizado independientemente de JavaScript, y muchos entornos de programación poseen la

capacidad de leer (convertir, parsear) y generar JSON.

Decimos que JSON es un string con un formato específico, lo que implica que si quiero crear un objeto usando la información de esta cadena, o transformar un objeto en un string JSON, existe un proceso de conversiòn Objeto-JSON y JSON-Objeto.

Convertir objetos a JSON

Con JSON.stringify podemos transformar un objeto JavaScript a un string en formato JSON. Es posible emplear el método de la siguiente manera:

```
const producto1 = { id: 2, producto: "Arroz" };
const enJSON    = JSON.stringify(producto1);

console.log(enJSON); // {"id":2,"producto":"Arroz"}
console.log(typeof producto1); // object
console.log(typeof enJSON);    // string

localStorage.setItem("producto1", enJSON);
// Se guarda {"id":2,"producto":"Arroz"}
```

Como podemos observar, contamos con un objeto global llamado JSON que nos permite realizar esta transformación usando el método stringify, el cual acepta un objeto como parámetro, y devuelve la forma de texto JSON equivalente. Este proceso se realiza cuando quiero almacenar la información de un objeto correctamente en storage, o deseo enviar los datos del objeto al servidor para ser procesados.

Convertir JSON a objetos

Con JSON.parse podemos transformar string en formato JSON, a objeto JavaScript. Es posible emplear el método de la siguiente manera:

```
const enJSON    = '{"id":2,"producto":"Arroz"}';
const producto1 = JSON.parse(enJSON);

console.log(typeof enJSON);      // string
console.log(typeof producto1);   // object
console.log(producto1.producto); // Arroz

const producto2 = JSON.parse(localStorage.getItem("producto1"));
console.log(producto2.id);   // 2
```

Como podemos observar, utilizamos el método parse cuando tenemos una cadena en en formato JSON, ya sea del storage o desde un variable, y necesitamos un objeto creado a partir de estos datos. El método recibe el string como parámetro, y devuelve el objeto JavaScript correspondiente. Este proceso se realiza cuando queremos recuperar información de un objeto almacenado en storage o para obtener datos del servidor con formato de texto plano.

Trabajando con storage y objetos personalizados

La pregunta es: ¿y por qué tengo que almacenar los objetos? Recordemos que las variables y estructuras de datos son elementos que existen en memoria, lo que implica que no son persistentes. En consecuencia, si los datos del usuario no se almacenan en algún medio, cuando se refresca la página la información generada se pierde, "reiniciando" la aplicación al estado inicial, es decir, empezamos de cero y todo procesamiento realizado previamente debe repetirse.

Como no interactuamos con servidores aún, storage es el medio que podemos utilizar para preservar datos del cliente. Esto también se realiza en aplicaciones modernas. Para entender el por qué, tengamos presente que es más "barato" (en términos de performance de la aplicación y costos) contar con información *prescindible* de uso recurrente almacenada en el cliente, que en el servidor. Si bien existe un límite de la

cantidad de información a guardar en storage (entre 5 y 10 MB, dependiendo del navegador), es un medio apropiado para conservar datos temporales o generados por el propio cliente antes enviarlos al servidor, pendientes de confirmación.

En la construcción de nuestra aplicación, podemos usar localStorage como recurso para almacenar la información del usuario y realizar simulaciones más complejas. Un caso posible es almacenar datos cargados por el usuario en storage. En el ejemplo, se detallan en un array de literales, pero la información podría venir de objetos instanciados usando entradas por prompt():

```
const productos = [{ id: 1,  producto: "Arroz", precio: 125 },
                   {  id: 2,  producto: "Fideo", precio: 70 },
                   {  id: 3,  producto: "Pan"  , precio: 50},
                   {  id: 4,  producto: "Flan" , precio: 100}];

const guardarLocal = (clave, valor) => { localStorage.setItem(clave, valor) };

//Almacenar producto por producto
for (const producto of productos) {
    guardarLocal(producto.id, JSON.stringify(producto));
}
// o almacenar array completo
guardarLocal("listaProductos", JSON.stringify(productos));
```

Luego, una vez cerrada la aplicación, se pueden recuperar los datos ya existentes sobre los objetos previamente creados de la siguiente manera:

```
class Producto {
    constructor(obj) {
        this.nombre   = obj.producto.toUpperCase();
        this.precio   = parseFloat(obj.precio);
    }
    sumaIva() {
        this.precio = this.precio * 1.21;
    }
```

```
}
//Obtenemos el listado de productos almacenado
const almacenados = JSON.parse(localStorage.getItem("listaProductos"));
const productos = [];
//Iteramos almacenados con for...of para transformar todos sus objetos a
tipo producto.
for (const objeto of almacenados)
    productos.push(new Producto(objeto));
//Ahora tenemos objetos productos y podemos usar sus métodos
for (const producto of productos)
    producto.sumaIva();
```

Hay que tener presente que cuando se parsean los objetos con JSON parse, y se necesita construirlos usando cierta clase, es necesario instanciar nuevos objetos de este tipo a partir de objetos obtenidos del parseo, como podemos ver en el ***for...of*** sobre el array cuyo identificador es almacenado.