UD7

Almacenamiento de datos en el lado cliente

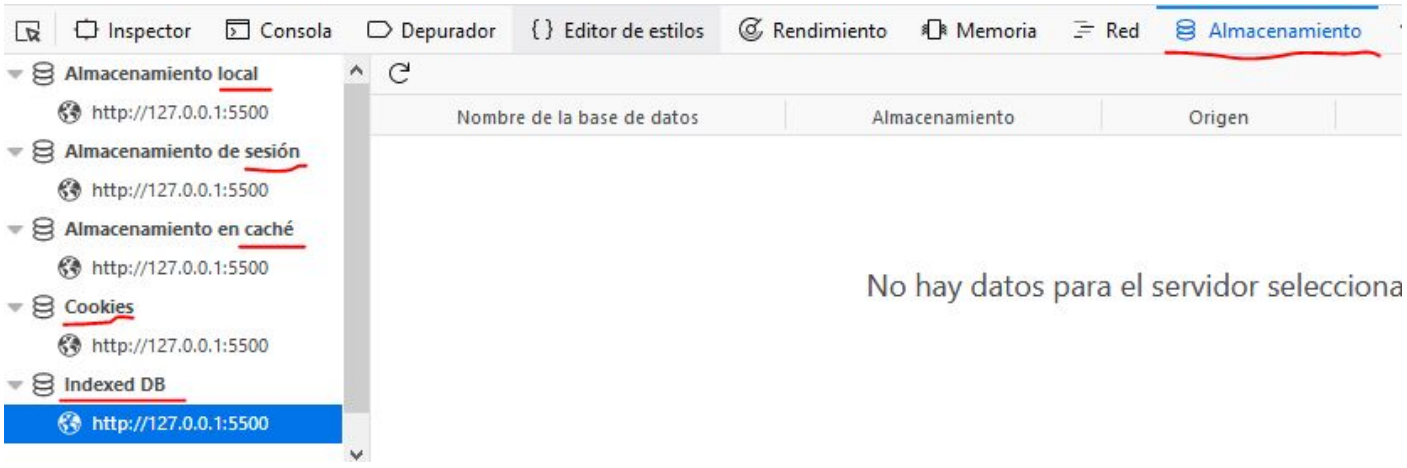
1. Introducción

Cuando construimos aplicaciones web es muy usual que en determinadas situaciones la implementación de ciertas características impliquen la **necesidad de almacenar datos** en el lado cliente. Algunos de estos escenario son bien conocidos como por ejemplo, tras loguearnos en un sistema almacenar durante un tiempo estimado un token de sesión, o cuando navegamos en un carrito de compra en la que debemos mantener los productos que se han añadido al carrito, o en un videojuego, mantener la puntuación, nivel, etc. sobre el jugador entre diferentes ventanas.

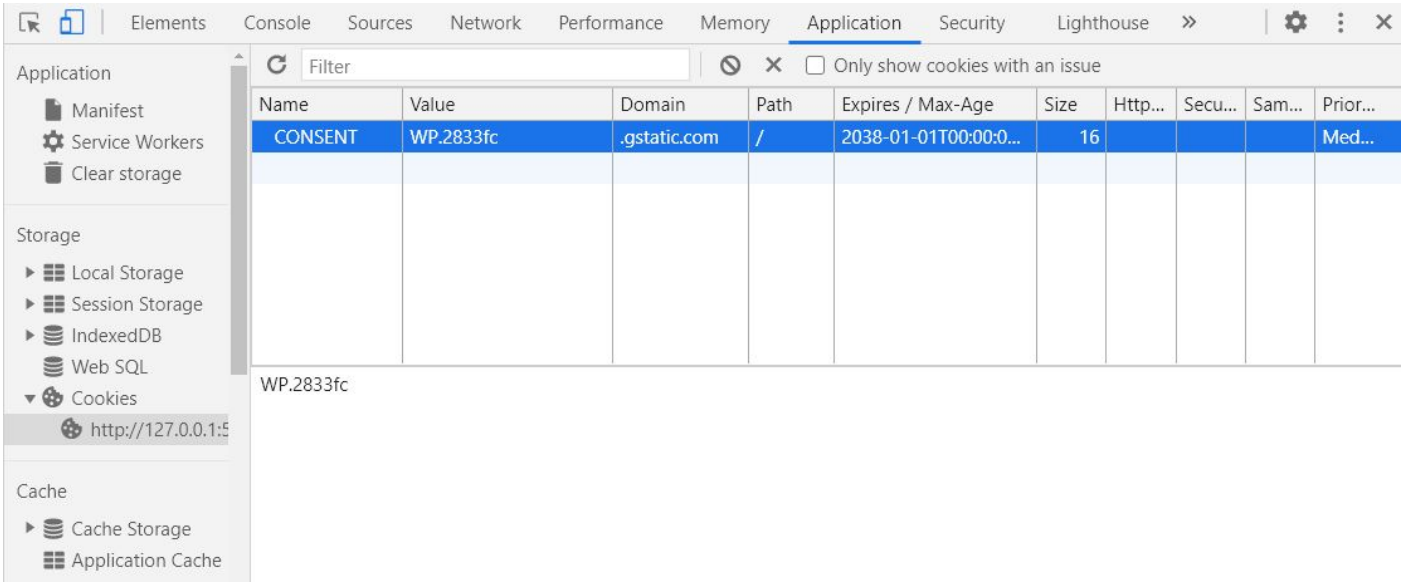
Bien, pues actualmente podemos distinguir diversas tecnologías que los navegadores nos ofrecen para el almacenamiento de datos en el lado cliente. Las tecnologías mencionadas son:

- Cookies
- Web Storage: Local Storage y Session Storage.
- IndexedDB
- Almacenamiento en caché (PWA)

En la siguiente pestaña de la ventana de desarrollador de Mozilla podemos ver dichos tipos de almacenamiento:



En Chrome las podemos encontrar en la pestaña “applications”, y en la barra lateral izquierda podremos visualizar los diferentes tipos de almacenamiento:



2. Cookies

¿Qué son las Cookies? Son datos almacenados en nuestros navegadores web en pequeños archivos de texto. Estos archivos de texto son asociados a cada navegador, es decir, puedo tener una serie de cookies guardadas en Chrome y otras en Firefox. Gracias a las cookies el navegador puede recordar información relativa a un usuario aunque se cierre el navegador o se desconecte del servidor. Por ejemplo, podemos guardar el nombre de un visitante de la página, el número de veces que ha entrado, para guardar los elementos guardados en el carrito de compra de una web, etc.

La información de las cookies se guarda en **forma de pares "nombre=valor"**: usuario = “Antonio”

- El **tamaño máximo** de cada cookie no puede exceder los **4096 bytes**.
- Un usuario no puede almacenar más de 300 cookies por dominio.

2.1 Operaciones básicas. Implementación.

Las cookies se utilizan en código gracias al objeto **“document.cookie”.** A continuación veremos cómo realizar una serie de operaciones sobre cookies.

2.1.1 Crear cookies

El formato de las cookies cuando las creamos puede ser diferente según cómo queramos configurarla:

Podemos crear una cookie simplemente con el par **“nombre = valor”**:

```
document.cookie = "nombre_cookie=valor_cookie;";
```

Con el parámetro **“expires”** podemos añadir una fecha de expiración de nuestra cookie. Por defecto las cookies son eliminadas con la pestaña o el navegador es cerrado a menos que le añadamos una fecha de expiración. La fecha debe de estar en formato UTC. Debemos de utilizar el siguiente formato:

```
document.cookie = "nombre_usuario=Juan; expires=Thu, 19 Jan 2018 13:45:00 UTC"
```

Con el parámetro **“path”** podemos añadir además la ruta de donde queremos que se almacene la cookie. Si no se utiliza, por defecto se almacenan en el directorio raíz.

```
document.cookie = "nombre_usuario=Juan; expires=Thu, 19 Jan 2018 13:45:00 UTC;
path=/";
```

2.1.2 Leer cookies

La lectura de cookies no es directa ya que se nos devuelven todas las cookies almacenadas:

```
console.log( document.cookie );
```

2.1.3 Modificar cookies

Para modificar una cookie tan solo debemos usar la misma “clave” para modificar su “valor”:

```
document.cookie = "nombre_usuario = Pedro"
```

2.1.4 Borrar cookies

Para borrar cookies tan solo debemos de hacer mención de su “clave” y asignarle una fecha ya expirada (da igual la fecha que sea, siempre que sea una fecha pasada), y así la cookie será eliminada. Es aconsejable incluir la ruta de la cookie para asegurarnos que estamos eliminando la cookie deseada:

```
document.cookie = "nombre_usuario=;  expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/;"
```

2.1.5 Entendiendo el concepto de path y dominios en las cookies.

Por defecto, y por seguridad, una cookie puede ser leída solo por páginas HTML que están alojadas en el mismo servidor web que la página que creó la cookie.

Por ejemplo, si tu tienes código javascript en “http://chimpance.webmonkey.com/comida/bananas/banana_pure.htm”

en la cual se pregunta a los usuarios por sus nombres, quizás en otra web necesitarías acceder a ese nombre de usuario, por ejemplo en la página **“home”** (http://chimpance.webmonkey.com/). Para permitir esto, deberías de configurar el “path” de la página correctamente. El “path” configura el directorio raíz desde el cual una cookie puede ser leída. Si quisiéramos configurar el path al directorio raíz de tus páginas web, dicha cookie debe de ser accesible en todas las otras páginas. En este caso deberías de asignar el **path=”/”** a tu cookie. Si quisiéramos que solo fuese accesible en el directorio “comida”, deberíamos asignar el **path=”/comida”**

Otra perspectiva sobre las cookies es que muchos sitios webs utilizan muchos **subdominios**. Por ejemplo, Webmonkey podría tener los subdominios: *chimpance.webmonkey.com* , *gorila.webmonkey.com y perezoso.webmonkey.com*. Por defecto, si creamos una cookie en una página del dominio “chimpance.webmonkey.com” solo podrán ser accedidas desde otras páginas del mismo dominio. Si quisieramos que todas las páginas del sitio *“webmonkey.com”* pudieran acceder a todas las cookies de todas las webs de todos los subdominios deberíamos de setear la propiedad **“domain=webmonkey.com”**.

Vamos a resumir todo lo anterios y configura una cookie creada en la página “http://chimpance.webmonkey.com/food/bananas/banana_pure.htm ” y que pueda ser accedido por cualquier página de “webmonkeys” tendríamos que configurar la cookie como:

```
function setCookie() {
  let  nombre = prompt("¿Cuál es tu nombre?", "");
  let  cookie = "cookie1=" +  nombre  + ";" ;
  let  cookie += "path=/;";
  let  cookie +=  "domain=webmonkey.com;";
  document.cookie = the_cookie;  }
```

2.2 Ejemplo de uso de cookies

En la web de la W3Schools dedicada a las cookies podemos ver una serie de ejemplos de funciones que nos facilitarán la vida a la hora de trabajar con cookies. En los siguiente métodos podemos ver como manejar a más alto nivel las operaciones de crear una cookie, obtener una cookie determinada y un método para comprobar si una determinada cookie existe o no.

➜ **Crear/configurar cookie:** Se creará una función en la que se le pase como argumento el campo nombre, valor y días de expiración:

```
function setCookie(cname, cvalue, exdays) {
  var d = new Date();
  d.setTime(d.getTime() + (exdays*24*60*60*1000));
  var expires = "expires=" + d.toUTCString();
  document.cookie = cname + "=" + cvalue + ";" + expires + ";path=/";
}
```

➜ **Obtener una cookie:** Se creará una función en la que se le pase como parámetro el nombre de la cookie a identificar:

```
function getCookie(cname) {
  var name = cname + "=";
  var decodedCookie = decodeURIComponent(document.cookie);
  var listado = decodedCookie.split(';');
  for(var i = 0; i < listado.length; i++) {
      var c = listado[i];
      while (c.charAt(0) == ' ') {
         c = c.substring(1);
      }
      if (c.indexOf(name) == 0) {
          return c.substring(name.length, c.length);
      }
  }
  return "";
}
```

3. Web Storage. Una alternativa a las cookies.

Web Storage fue una característica introducida en HTML5. Web Storage nos permite guardar información localmente en el equipo del usuario, al igual que hacen las cookies pero con algunas diferencias que mejoran en líneas generales al uso de cookies.

Algunas de las características a destacar de los Web Storage son:

- La información guardada en el Web Storage no es enviado a el servidor por cada petición HTTP (al contrario que con Cookies) lo que aligera la cantidad de información de tráfico enviada entre el cliente y el servidor.
- La capacidad de información guardada en los Web Storage es aproximadamente de 10 MB (al menos 5 MB). Con Cookies la información máxima almacenable es de 4 KB.

Resumiendo, haciendo uso de las WebStorage tenemos más capacidad de almacenamiento sin perjudicar el rendimiento de una web (en las comunicaciones con el servidor).

Nota: Soporte de navegadores
Prácticamente todos los navegadores actuales dan soporte al uso de WebStorage. Podemos comprobarlo en la siguiente web la cual nos informa sobre qué podemos o no usar en los diferente navegadores:
Can I use ?

3.1. Tipos de Web Storage.

Existen 2 tipos de Web Storage que pueden ser usados para almacenar datos de forma local:

- **Local Storage:** Los datos almacenados no tienen fecha de expiración. Los datos son guardados en la máquina del cliente hasta que sean borrados de forma implícita, incluso seguirán disponibles si el navegador es cerrado y reabierto.

- **Session Storage:** Este tipo de almacenamiento de datos en el lado cliente se almacena solamente durante la sesión actual. Una vez que el navegador sea cerrado

3.2. Operaciones básicas. Implementación.

Es importante mencionar que para los objetos **localStorate** como los **sessionStorage** se utiliza los **mismo métodos para operaciones básicas**.

3.2.1 Comprobar si está soportado

Antes de nada debemos añadir en nuestro código una comprobación de si está o no soportado por el navegador el uso de local o session storage

```
    if (window.localStorage) {
            // Navegador soporta localStorage    o    session storage
            // Vamos para adelante:
        } else
        {
            // No hay soporte de localStorage
        }
```

3.2.2 Crear un almacenamiento de un dato (setItem)

```
        localStorage.setItem('Nombre', 'valor');
```

3.2.3 Leer un dato (getItem)

```
        localStorage.getItem("Nombre");
```

3.2.4 Eliminar un dato (removeItem)

```
        localStorage.removeItem("Nombre");
```

3.2.5. Eliminar todos los datos almacenados (clear)

```
        localStorage.clear();
```

3.2.6 Obtener el número de elementos que hay almacenados (length):

```
        localStorage.length;
```

Importante: Seguridad gracias al dominio
Tiene sentido que las cookies y los web storage se configuren para un determinado dominio. Imagínate el grado de inseguridad que provocaría que desde unas páginas podamos ver las cookies de otras páginas.

3.3. Almacenando objetos en WebStorage

En WebStorage, en teoría, solo se puede almacenar valores primitivos pero existe una forma de almacenar objetos completos mediante la serialización. Gracias a los métodos **JSON.parse()** y **JSON.stringify()** podemos almacenar y recuperar objetos completos almacenados mediante WebStorage.

```
var persona = {

  Nombre: 'Javier',
  Edad: 24,
  Genero: 'Masculino',
  Nacionalidad: 'Española'
}

localStorage.setItem('persona', JSON.stringify(persona)); //Almacenar objeto
JSON.parse(localStorage.getItem('persona')); //Recuperar el objeto
```

Nota: **"Ojo con la Serialización"**
No se debe de hacer un uso indiscriminado de la serialización debido a que suele ser una tarea que consume una cantidad importante de tiempo de ejecución.

¡¡¡Este curso 2020-2021 no estudiaremos indexedDB!!!

4. IndexedDB

IndexedDB es un sistema de base de datos transaccionales del lado cliente que nos permite construir aplicaciones web capaces de **almacenar considerables colecciones de datos en el lado cliente**. Hasta ahora habíamos visto WebStorage como una solución a esta necesidad, pero nos encontramos que el almacenamiento se basa en simples datos siguiendo una estructura de **pares clave/valor**. IndexedDB soporta el **almacenamiento de "estructuras de datos" en el lado cliente**, también siguiendo la estructura de **pares clave/valor** pero permitiendo a los desarrolladores construir aplicaciones web más complejas ya que en los valores podemos insertar **objetos, strings numbers, arrays y dates**.

Permite hacer **versiones** de nuestra base de datos, utilizar **transacciones** para mantener la integridad de nuestra base de datos, y nos permite crear **varias bases de datos** para el mismo sitio pero siempre manteniendo la **restricción de un mismo dominio** para que dichas bases de datos no puedan ser accedidas desde otro sitios.

4.1. Comprobar que IndexedDB esté soportado

Para comprobar si nuestro navegador tiene soporte para este tipo de bases de datos debemos de comprobarlos con el siguiente código:

```
if (!('indexedDB' in window)) {
  console.log("Este navegador NO soporta IndexedDB”);
  return;
}
```

Os dejo aquí la línea en la que figura una serie de objetos que representan a la base de datos indexedDB en distintos navegadores:

```
let indexedDB = window.indexedDB || window.mozIndexedDB || window.webkitIndexedDB ||
window.msIndexedDB
if (!indexedDB)
   console.log(“No soportada”);
```

4.2. Creando una base de datos IndexedDB

Con IndexedDb se puede crear múltiples bases de datos para la misma web siempre que tengan nombres diferentes, pero normalmente existe una sola base de datos por aplicación web. Para abrir/crear una base de datos debemos usar el método **“open”**. En el siguiente ejemplo podemos ver la sintaxis:

```
window.indexedDB.open(nombre, version, upgradeCallback)
```

Como vemos los parámetros deben ser el nombre de la base de datos, como mínimo, aunque también podemos añadir la número de la versión de la base de datos y también opcionalmente una función callback.

Veamos un ejemplo completo:

```
if (!('indexedDB' in window)) {
      console.log("Este navegador NO soporta IndexedDB");
      return;
  }
  var openDB = indexedDB.open('test-db1', 1);
```

Cuando abrimos una base de datos debemos de configurar una serie de eventos que podemos utilizar para realizar ciertas tareas dependiendo del éxito o tipo de creación de la base de datos. Estos eventos son:

- **onupgradeneeded**: Se llamará cuando la base de datos ha sido creada por primera vez, o su número de versión haya cambiado respecto a la versión ya existente. Normalmente, se utiliza este evento para la generación/eliminación de los “almacenes de datos” (objet stores).
- **onsuccess**: Se llamará cuando la base de datos ha sido creada o abierta correctamente.
- **onerror**: Se llamará cuando la base de datos no haya podido ser creada o abierta correctamente.

4.3. Creando Object Stores

Los **Object Stores** vienen a ser un tipo de almacén de datos que podríamos compararlos a las **“tablas”** en bases de datos relacionales. Una base de datos de tipo indexedDB debería de ser bien construida creando un object store por cada tipo de objeto que se quiera representar, pero no es obligatorio. La sintaxis básica de método para crear Object Stores es:

```
upgradeDb.createObjectStore('Nombre_Object_store', opciones);
```

Ejemplos de creación de **Object Stores**:

- Creando un object store y generando su clave primaria de forma autoincremental:

```
upgradeDb.createObjectStore('notes', {autoIncrement:true});
```

- Creando un object store con una determinada clave primaria autoincremental:

```
upgradeDb.createObjectStore('logs', {keyPath: 'id'});
```

- Creando un object store con una determinada clave primaria autoincremental:

```
upgradeDb.createObjectStore('logs', {keyPath: 'id', autoIncrement:true});
```

Siguiendo el hilo del código de ejemplo anterior vamos a continuarlo y vamos a crear nuestro primer Object Store, y debemos de fijarnos si este ya existe o no, para no ser creado por duplicado haciendo uso del método **“db.objectStoreNames.contains(“nombre_object_store”)”** :

```
let db;
openedDB.onupgradeneeded = function(e) {
    db = e.target.result;

     if (!db.objectStoreNames.contains('store'))
        db.createObjectStore('store', { autoIncrement: true })

     if (!db.objectStoreNames.contains('objectStoreName2'))
        db.createObjectStore('objectStoreName2', { keyPath: "nombre" })
}

openedDB.onsuccess = function(e)
{
    db = e.target.result;
    alert("Base de datos creada correctamente");
}
```

4.4. Creando índices (index) (NO CAE)

Índices (index) son un tipo de objetos usados para “capturar/coger” datos de un “objet store” haciendo uso de una “propiedad específica”. Los índices nos permiten realizar búsquedas de elementos almacenados ateniéndonos a campos diferentes a las claves primarias. Su uso se debe usar sólo en aquellas situaciones que vayan a ser realmente utilizados ya que cada vez que se realiza una modificación de un “object store”, la base de datos debe de reconstruir su mapa de índices, y esto conlleva una operación costosa. Podemos incluso limitar si sus valores van a ser únicos o no. Veamos un ejemplo:

```
if (!db.objectStoreNames.contains('people')) {
      var peopleOS = db.createObjectStore('people', {keyPath: 'email'});
      peopleOS.createIndex('genderIndex', 'gender', {unique: false});
      peopleOS.createIndex('ssnIndex', 'ssn', {unique: true});
      }
```

El primer argumento es el “**nombre**” del índice. El segundo es el **nombre del dato/campo** que vas a representar con este índice. La mayoría de las veces se usará el mismo valor para el primer y el segundo parámetro. Para el tercer parámetro es la restricción sobre los valores a obtener.

4.5. Trabajando con datos

Todas las operaciones que se realizan con IndexedDB son llevadas a cabo de forma “asíncrona” y se utilizan **“transactions” (transacciones)**. Las transacciones son una forma (un wrapper) de realizar operaciones que nos garantizarán ciertas medidas de seguridad como por ejemplo en el caso de que algo fuera mal, cualquier cambio puede ser deshecho. Las transacciones se realizarán sobre uno o varios “object store”.

Para crear un **“transacción”** debemos darle el nombre del object store sobre el que queremos realizar la operación de lectura, escritura, borrado, etc, de datos y el tipo de permiso con el que queremos realizar la operación. Estos valores pueden ser **“readonly”** o **“readwrite”**:

```
var tx = db.transaction('store', 'readonly');
var store = tx.objectStore('store');
```

4.5.1 Creando datos

Para crear datos en la base de datos usaremos el método “add”:

```
someObjectStore.add(data, optionalKey);
```

Veamos un poco de código:

```
  var tx = db.transaction('store', 'readwrite');
  var store = tx.objectStore('store');
  var item = {
```

```
    name: 'sandwich',
    price: 4.99,
    description: 'A very tasty sandwich',
    created: new Date().getTime()
};
store.add(item);
```

4.5.2 Leyendo datos

Para leer datos en la base de datos usaremos el método “get”:

```
someObjectStore.get(optionalKey);
```

Veamos un poco de código:

```
var tx = db.transaction( 'store' , 'readonly');
var store = tx.objectStore('store');
console.log(store.get('sandwich'));
```

Importante: Leyendo datos que no existen
Si tratas de obtener un objeto que no existe, la operación se ejecuta sin errores, pero el resultado será de tipo **“undefined”.**

4.5.3 Actualizando datos

Para actualizar datos en la base de datos usaremos el método “put”:

```
someObjectStore.put(data, optionalKey);
```

Veamos un poco de código:

```
var tx = db.transaction('store', 'readwrite');
var store = tx.objectStore('store');
var item = {
  name: 'sandwich',
  price: 99.99,
  description: 'A very tasty, but quite expensive, sandwich',
  created: new Date().getTime()
};
store.put(item);
```

4.5.4. Borrando datos

Para borrar datos en la base de datos usaremos el método “delete”:

```
someObjectStore.delete(primaryKey);
```

Veamos un poco de código:

```
var tx = db.transaction('store', 'readwrite');
var store = tx.objectStore('store');
store.delete(key);
```

4.5. Recorriendo los datos guardados en los Object Stores

Para recorrer los datos de nuestros Object Stores es haciendo uso de cursores. Un **cursor** es un puntero sobre nuestro Object Store el cual podremos utilizar para iterar por todos los elementos del ObjectStore, y en cada iteración poder acceder a todos los valores, atributos de cada uno de ellos.

La sintaxis del cursor básicamente es la siguiente:

```
var request = ObjectStore.openCursor();
var request = ObjectStore.openCursor(query);
```

En el primer caso, sin usar ningún parámetro podremos iterar por todos los objetos de nuestro ObjectStore. En el segundo caso, utilizaremos como parámetro un objeto de tipo **IDBKeyRange,** con el cual podremos filtrar los resultados que nos devolverá el cursor. Para conocer más sobre cómo configurar este tipo de objetos podemos visitar el siguiente enlace: "IDBKeyRange". Veamos algunos ejemplos:

```
let db = openedDB.result;
      if (db)
      {
            let contenido="";
            let tx = db.transaction('objectStoreName', "readonly")
            let store = tx.objectStore('objectStoreName')
            let request = store.openCursor(IDBKeyRange.bound("A", "F"));
            request.onsuccess = function(event) {
               var cursor = event.target.result;
               if(cursor) {
                  console.log(cursor.value.firstname);
                  cursor.continue();  //MUY IMPORTANTE
               } else {
                  // no more results
                                    }
               };
      }
```

Como podrá ver para iterar por los resultados tenemos que utilizar un nuevo método de los cursores: **cursor.continue()**.

5. HTML5 Manifest Caché (NO ESTUDIAR PARA EXAMEN)

Normalmente la mayoría de los sitios web están implementados para trabajar solamente en modo “online”, es decir, con conectividad, pero ¿Qué pasa si nuestra web se queda sin conexión?. El **cache manifest** es una de las novedades de html5 que nos ayuda a la hora de diseñar el comportamiento de nuestro sitio web cuando funciona sin conexión. Este se encarga de almacenar en la memoria del navegador todos aquellos recursos de nuestra web que quieran ser mantenidos/visibles/accesibles en el caso de que la web se quedara sin conectividad.

Algunas de las ventajas de usar **HTML5 application cache** son:

- **Navegación Offline:** Los usuarios pueden usar la web incluso cuando esta está offline debido por ejemplo a una breve desconexión.
- **Mejora el rendimiento:** Todos aquellos recursos que se quedan cacheados influye enormemente en el rendimiento a la hora de cargar la web desde el servidor, al no ser necesario cargar aquellos recursos cacheados.
- **Reduce el número de HTTP request y la carga de trabajo del servidor:** debido a que muchos de los recursos quedan cacheados y no tenemos que recargarlos.

Para usar HTML5 application cache debemos de crear un archivo con el nombre que deseemos pero normalmente con la extensión .cache o .appcache. En este archivo adjuntar los recursos a cachear en 3 tipos de categorías (CACHE, NETWORK y FALLBACK), veamos un ejemplo:

```
CACHE MANIFEST

      # v1.0 : 10-08-2014

      CACHE:

      # Páginas o recursos que van a ser cacheados para ser accesibles cuando la
      # web esté offline

      index.html

      css/theme.css

      js/jquery.min.js

      js/default.js

      /favicon.ico

      images/logo.png


      NETWORK:

      # Los archivos bajo NETWORK son aquellos que no van a ser cacheados.
      login.php
```

```
FALLBACK:

/ /offline.html

# FALLBACK es la tercera categoría y representa aquellos archivos que estén
# por debajo del directorio indicado (en este caso el raíz de tu página web) y
# que además no estén cacheados, serán sustituidos por /offline.html
```

Una vez que tengas el archivo construido debemos usarlo como cabecera de nuestro HTML:

```
<html lang="en" manifest="manifiesto.cache">
```

6. Enlaces

- **Referencia sobre cookies:**
  https://www.w3schools.com/js/js_cookies.asp

  Videos Ada Lovelace sobre cookies: Video 1 Video 2 Video 3

- **Referencia sobre Web Storage:**
  https://codeburst.io/using-html5-web-storage-a450294484bb
  https://www.w3schools.com/HTML/html5_webstorage.asp

  Videos Ada Lovelace sobre cookies: Video 1 Video 2

- **Referencias sobre indexedDB:**
  Working with indexedDB DesarrolladoresWeb Tutorial bastante Pro

  VideoTutorial sobre indexedDb en 3 vídeos (muy bien explicado):
  Enlace 1 Enlace 2 Enlace 3

- Enlace de referencia sobre todos los tipos de almacenamiento web (muy completo):
  Mozilla Web Store

  https://github.com/jakearchibald/idb ➜ Librería usada para indexedDB con Promises

- **Referencias sobre Web Caché:**
  https://iagolast.wordpress.com/2012/11/21/html5-cache-manifest/
  https://www.youtube.com/watch?v=q8T6-bS6liM ➜ Español (avance un poco el video)
  https://youtu.be/r_2ZaBNaETo