JavaScript

Almacenamiento de datos en el cliente

Contenido

1. Introducción.

Uno de los pilares de la personalización se encuentra en el concepto de sesión y en la habilidad de almacenar datos del usuario que utiliza el sitio web en el mismo navegador cliente con el fin de mejorar la experiencia del usuario. Si la información pertenece al usuario debe estar en la localización del usuario (es decir en el navegador cliente). A continuación, veremos las diferentes opciones de almacenamiento de datos en los navegadores clientes.

Los navegadores web modernos admiten varias formas para que los sitios web almacenen datos en la computadora del usuario, con el permiso del usuario, y luego los recuperen cuando sea necesario. Esto te permite conservar los datos para el almacenamiento a largo plazo, guardar sitios o documentos para su uso sin conexión, conservar la configuración específica del usuario para tu sitio y más.

Almacenar datos en el lado del cliente tiene muchos usos distintos, como:

- Personalizar las preferencias del sitio (por ejemplo, mostrar la elección de un usuario de artilugios personalizados, combinación de colores o tamaño del tipo de letra).
- Persistencia de la actividad anterior del sitio (por ejemplo, almacenar el contenido de un carrito de compras de una sesión anterior, recordar si un usuario inició sesión anteriormente).
- Guardar datos y activos localmente para que un sitio sea más rápido (y potencialmente menos costoso) de descargar o se pueda usar sin una conexión de red.
- Guardar documentos generados por aplicaciones web localmente para usarlos sin conexión

2. Las cookies

Mecanismo utilizado para el intercambio de información en Internet. Pero HTTP es un protocolo sin estado, por tanto, el servidor web administrará cada nueva petición HTTP de manera independiente y sin retener ningún tipo de información, como, por ejemplo, los valores de las variables y campos que se han utilizado en cada solicitud.

Por supuesto, esto disminuye la capacidad de las aplicaciones web por lo que la construcción de aplicaciones que necesiten mantener el estado entre peticiones es una tarea más complicada. Es por ello que se implementan mecanismos para lograr la persistencia de los datos a partir de sesiones.

Cookie: pequeña cantidad de datos **almacenada**, en forma de ficheros de texto, **en el cliente** en forma de pares clave/valor, con una fecha o un tiempo de expiración (es opcional, si no se especifica se borra al cerrar la sesión) y relacionada con un dominio

determinado. **El navegador puede envíar la cookie al servidor cuyo nombre de dominio sea igual a ese atributo con cada petición HTTP que se realice.**

2.1 Gestión de cookies en Javascript

Las cookies son almacenadas en el ordenador del cliente en formato ***clave=valor***. Un ejemplo podría ser *username=Mario* para recordar el nombre de usuario o *color=black* para recordar el color preferido de un visitante. En caso de que no pongamos una fecha de expiración, la cookie se eliminará al cerrar el navegador, así que también podemos especificar la fecha en que el navegador del usuario la borrará.

Creación de una cookie Javascript:

```
document.cookie="usuario=Mario; expires=Mon, 5 Feb 2024 15:00:00 UTC";
```

El navegador almacenaría el nombre de usuario y la fecha de expiración sería el lunes 5 de febrero de 2024 a las 15:00 horas. En caso de querer modificarla, haríamos lo mismo que al crearla, pero con nuevos datos. Para eliminarla, le ponemos una fecha que ya haya pasado:

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

o también:

```
document.cookie = "username=; max-age=0";
```

Si queremos crear varias cookies, tenemos que hacer este paso una vez para cada una. Por ejemplo:

```
document.cookie = "usuario=Pepe; max-age=3600; path=/;";
document.cookie = "color_favorito=azul";
```

Otros parámetros para las cookies:

- Campo **path=<ruta>**. Opcional. Establece la ruta para la cual la cookie es válida. Si no se especifica ningún valor, la cookie será válida para la ruta de la página actual. La ruta debe ser **absoluta**.

- Campo **max-age=<segundos>**. Opcional. Establece una duración máxima en segundos. Tiene preferencia sobre expires. Si no se especifica ninguno de los dos se creará una sesión cookie. Si es max-age = 0 la cookie se elimina.

- Campo **domain=<dominio>**. Opcional. Dentro del dominio actual, subdominio para el que la cookie es válida. El valor predeterminado es el subdominio actual. Se puede establecer, por ejemplo, el valor ***domain=.miweb.com*** para una cookie que sea válida en cualquier subdominio (*nota el punto delante del nombre del dominio*). Por motivos de seguridad, los navegadores no permiten crear cookies para dominios diferentes al que crea la cookie (*same-origin policy*).

- Campo **secure**. Opcional. Parámetro sin valor. Si está presente la cookie sólo será enviada en conexiones encriptadas (por ejemplo, mediante protocolo HTTPS).

- Campo **samesite** Este atributo impide al navegador enviar esta cookie a través de peticiones cross-site.

Cada vez que se crea una nueva cookie, no se sobrescriben las cookies anteriores, sino que **la nueva se añade a la colección de cookies del documento**.

Recuerda que las cookies se envían en las cabeceras HTTP y, por tanto, deben estar correctamente codificadas. Puedes utilizar ***encodeURIComponent()***, acostúmbrate a utilizarlo siempre para evitarte sorpresas:

```
var testValue = “Hola mundo!”;

document.cookie = “testcookie=” + encodeURIComponent (testvalue);
```

Métodos para la gestión de cookies:

Crear una cookie:

Para crear una cookie necesitaremos, al menos, tres campos: el nombre del valor, el valor de la cookie y la fecha de expiración.

Como hemos visto antes, asignaremos la cookie concatenando el nombre del campo de la cookie con un “=” y con el valor de dicha cookie. Si vamos a añadir varios campos seguidos a una cookie, deberemos separar los distintos campos con **“;”**.

Para asignar un tiempo de expiración para la cookie, además de lo realizado anteriormente, habrá que concatenar este tiempo. Para ello, podremos usar **`“max-age=”`** y concatenar el número de segundos que durará la cookie. Otra forma es usar **`“expires=”`** y asignarle una fecha ***que aún no haya pasado*** en formato UTC (formato internacional). Para ello tendremos que ayudarnos de los métodos del objeto **`Date`** de JavaScript como **`setTime(), getTime(), toUTCString()`**….

Obtener una cookie:

Lo que hace esta función es separar el valor de las cookies por puntos y comas y buscar la clave y valor correspondiente. Si no encuentra nada, devuelve un string vacío.

```
function obtenerCookie(clave) {
    var name = clave + "=";
```

```
    var ca = document.cookie.split(';'); // Obtenemos los campos de la cookie
    for(var i=0; i<ca.length; i++) {
        var c = ca[i];
        while (c.charAt(0)==' ')
            c = c.substring(1); // Eliminamos los espacios en blanco
        if (c.indexOf(name) == 0)
            return c.substring(name.length,c.length);
    }
    return "";
}
```

Comprobar si una cookie ha sido creada:

Esta función comprueba si una cookie existe o ha sido creada previamente mediante el uso de la función obtenerCookie(). Va a recibir como parámetro una clave.

```
function comprobarCookie(clave) {
    var clave = obtenerCookie(clave);
    if (clave != "") {
        // La cookie existe.
    }
    else {
        // La cookie no existe.
    }
}
```

Problemas con las cookies:

- Cada navegador tendrá sus propias cookies.
- No diferencian entre usuarios que utilicen el mismo navegador en una misma sesión.
- Son vulnerables a los ***sniffers*** (programas que pueden leer el contenido de peticiones y respuestas HTTP) debido a que estas se realizan en texto plano.
- Pueden ser modificadas en el cliente, lo cual podría aprovechar vulnerabilidades del servidor.

3. La especificación web storage de la W3C

El almacenamiento web ha cobrado más fuerza con la especificación HTML 5. HTML 5 incluye dos nuevos objetos para el almacenamiento de datos en el cliente: los ***sessionStorage*** y los ***localStorage***.

Una de las principales **diferencias con las cookies** está en que el contenido de las peticiones HTML 5 NO se envían en cada petición HTTP. De hecho, la información solo podrá ser accedida desde el lado del cliente por lo que es posible almacenar gran cantidad de información sin afectar el rendimiento de la aplicación web.

Algunos escenarios de implementación y uso son los siguientes:

- **Almacenamiento de datos**. Los datos son almacenados en el cliente y pueden ser pasados al servidor por intervalos, en lugar de en tiempo real.
- **Utilización fuera de línea**. Como consecuencia de que la relación entre los datos almacenados y el navegador no se pierden entre sesiones (*para localStorage*).
- **Mejora del rendimiento**. Podemos almacenar datos estáticos (por ejemplo, imágenes en codificación Base64) que no serán nuevamente obtenidos mediante peticiones al servidor. Podemos, además, guardar y recuperar objetos usando **`JSON.stringify()`** y **`JSON.parse()`**.
- En el caso del *sessionStorage* **no existe relación entre lo almacenado en diferentes pestañas o ventanas** de un mismo navegador.
- Con *sessionStorage* existe la seguridad de que **los datos serán borrados una vez termine la sesión de la ventana** que lo ha utilizado.

Las variables locales están asociadas a **protocolo**, **dominio** y **puerto.** Un programa solo puede acceder a propiedades de local/sessionStorage creadas por otros programas cargados del mismo servidor.

Estas propiedades se basan en ***Same origin policy***. Es una medida importante de seguridad para scripts en la parte cliente (casi siempre JavaScript). Previene que un documento o script cargado en un "*origen*" pueda cargarse o modificar propiedades del documento desde un "*origen*" diferente. Se trata de uno de los conceptos de seguridad más importantes de los navegadores modernos. Aportan:

- **Seguridad**: un programa solo confía en programas del mismo servidor.
- **Modularidad**: cada servidor tiene un espacio de nombres diferente.

3.1 sessionStorage

Las aplicaciones web pueden agregar información a este atributo el cual estará accesible durante toda la sesión. El objeto *sessionStorage* se instancia **por sesión y ventana**, por lo que dos pestañas del navegador abiertas al mismo tiempo y para un mismo sitio web pueden tener información distinta. Al cerrar la sesión se pierde la información.

Métodos del objeto SessionStorage

- Agregar un nuevo par clave/valor:
```
sessionStorage.setItem("maleta","1");
```
- Obtener el valor en base a la clave:
```
var item = sessionStorage.getItem("maleta");
```
- Borrar el par clave/valor. Puede ser llamado con la clave o con la posición del elemento a eliminar:

```
var item = sessionStorage.removeItem("maleta");
var item = sessionStorage.removeItem(1);
```

- El método "clear()" borra todos los elementos de la lista *sessionStorage*:
  `sessionStorage.clear();`

- Podemos usar otros atributos, por ejemplo, *length* para conocer la cantidad de elementos almacenados.
  `sessionStorage.length;`

3.2 localStorage

Está diseñado para los datos que se extienden a lo largo de múltiples ventanas y múltiples sesiones. Está orientado a las aplicaciones que necesitan mantener gran cantidad de información del usuario (en el orden de los megabytes), principalmente por motivos de rendimiento.

Se usa con las mismas propiedades que *sessionStorage*.

Añadir método key()

Ejercicio: Mostrar en la aplicación web la cantidad de veces que se ha visitado una página.

```
if (!localStorage.cuenta)
        localStorage.cuenta = 0;
      // También: localStorage.cuenta =(localStorage.cuenta || 0);
    else {
        localStorage.cuenta = parseInt(localStorage.cuenta) + 1;
        document.getElementById("cuenta").textContent = localStorage.cuenta;
    }
```

Un aspecto importante es que JavaScript realiza la conversión automática de tipo de dato numérico a cadena de caracteres. Por ello, al recuperar el valor almacenado es necesario transformarlo en tipo numérico.

4. Webs Progresivas

4.1 ¿Qué es una Web Progresiva?

El término "Aplicación web progresiva" no es un nombre formal u oficial. Solo es una abreviatura utilizada para el concepto de crear una aplicación **flexible y adaptable** utilizando solo tecnologías web.
Las PWA son **aplicaciones web** desarrolladas con una serie de tecnologías específicas y patrones estándar que les permiten aprovechar las funciones de las aplicaciones nativas y

web. Por ejemplo, las aplicaciones web son más fáciles de detectar que las aplicaciones nativas; es mucho más fácil y rápido visitar un sitio web que instalar una aplicación, y también puedes compartir aplicaciones web simplemente enviando un enlace.

Por otro lado, las **aplicaciones nativas** están mejor integradas con el sistema operativo y, por lo tanto, ofrecen una experiencia más fluida para los usuarios. Puedes instalar una aplicación nativa para que funcione sin conexión, y a los usuarios les encanta tocar sus íconos para acceder fácilmente a sus aplicaciones favoritas, en lugar de navegar a través de un navegador.

Las PWA brindan la capacidad de crear aplicaciones web que pueden disfrutar de estas mismas ventajas de las aplicaciones nativas.

No es un concepto completamente nuevo; estas ideas se han revisado muchas veces en la plataforma web con varios enfoques en el pasado. La mejora progresiva y el diseño adaptable ya te permiten crear sitios web compatibles con dispositivos móviles.

Sin embargo, las PWA brindan todo esto y más sin perder ninguna de las características existentes que hacen que la web sea excelente.

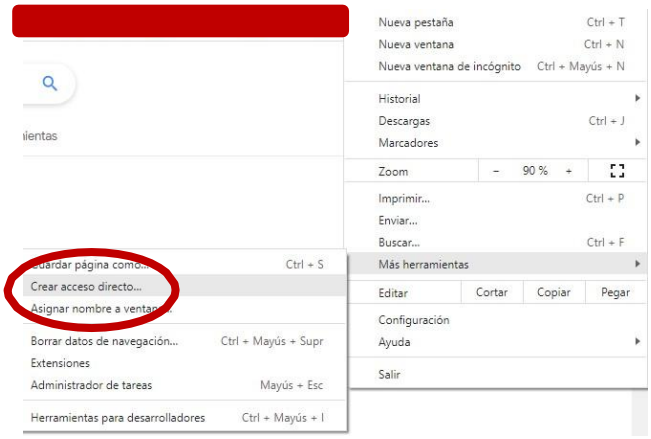
4.2 ¿Qué hace que una aplicación sea una PWA?

Como dijimos anteriormente, las PWA no se crean con una sola tecnología. Representan una nueva filosofía para la creación de aplicaciones web, que incluye algunos patrones específicos, API y otras características. A primera vista, *no* es tan obvio si una aplicación web es una PWA o no. Una aplicación se podría considerar una PWA cuando cumple con ciertos requisitos o implementa un conjunto de características determinadas: funciona sin conexión, es instalable, es fácil de sincronizar, puede enviar notificaciones automáticas, etc.

Además, existen herramientas para medir qué tan completa (como porcentaje) es una aplicación web, como **Lighthouse**. Al implementar varias ventajas tecnológicas, podemos hacer que una aplicación sea más progresiva, y así terminar con una puntuación de Lighthouse más alta. Pero este es solo un indicador aproximado.

Hay algunos principios clave que una aplicación web debe tratar de observar para ser identificada como PWA. Estos deben ser:

- **Detectable**, por lo que el contenido se puede encontrar a través de motores de búsqueda.
- **Instalable**, por lo que puede estar disponible en la pantalla de inicio del dispositivo o en el lanzador de aplicaciones. Para “instalar” una aplicación web podremos crear un acceso directo en el sistema para que el usuario acceda sin tener que introducir la URL. En Google Chrome, por ejemplo se hace mediante el menú – Más herramientas – Crear acceso directo.



- **Enlazable**, para que puedas compartirla simplemente enviando una URL.
- **Independiente de la red**, por lo que **funciona sin conexión** o con una deficiente conexión de red.
- **Progresiva**, por lo que todavía se puede utilizar en un nivel básico en los navegadores más antiguos, pero completamente funcional en los más recientes.
- **Reconectable**, por lo que puede **enviar notificaciones** cuando haya contenido nuevo disponible.
- **Adaptable**, por lo tanto, se puede utilizar en cualquier dispositivo con pantalla y navegador: teléfonos móviles, tabletas, computadoras portátiles, televisores, refrigeradores, etc.
- **Segura** por lo que las conexiones entre el usuario, la aplicación y tu servidor están protegidos contra terceros que intenten acceder a datos sensibles.

Ofrecer estas funciones y hacer uso de todas las ventajas que ofrecen las aplicaciones web puede crear una oferta atractiva y altamente flexible para tus usuarios y clientes.

Nosotros nos vamos a centrar en conseguir dos objetivos:

- Conseguir que nuestra aplicación web funcione sin conexión.
- Poder enviar notificaciones cuando haya cambios.

4.3 Ventajas de una PWA

Los beneficios son enormes. Por ejemplo:

- Una disminución en los tiempos de carga después de la instalación de la aplicación, gracias al **almacenamiento en caché** con el **service workers**, además de ahorrar un valioso ancho de banda y tiempo. Los PWAs tienen una carga casi instantánea (a partir de la segunda visita).
- La capacidad de actualizar solo el contenido que ha cambiado cuando hay disponible una actualización de la aplicación. En contraste, con una aplicación nativa, incluso la más mínima modificación puede obligar al usuario a descargar la aplicación completa nuevamente.

- Una apariencia que está más integrada con la plataforma nativa: íconos de aplicaciones en la pantalla de inicio o el lanzador de aplicaciones, aplicaciones que se ejecutan automáticamente en modo de pantalla completa, etc.
- Reconectable para interactuar con los usuarios mediante el uso de notificaciones del sistema y mensajes **push**, lo cual genera usuarios más comprometidos y mejores tasas de conversión.

Aquí tienes algunos estudios hechos sobre el aumento de uso de webs progresivas.

5. Saber si estamos conectados o no (Onlne / Offline)

Antes de almacenar datos, es posible que desee saber si el usuario está en línea o no. Esto puede ser útil, por ejemplo, para decidir si almacenar un valor localmente (lado del cliente) o enviarlo al servidor.

Podemos comprobar si nuestra aplicación web tiene conexión a Internet con **`navigator.onLine`**. Además, tenemos los eventos **offline** y **online**.

Según la información que aparece en MDN, no todos los navegadores han implementado esta propiedad de la misma forma. En general, Chrome y Safari devuelven true si el navegador se puede conectar a una red de área local (LAN) o a un router, lo cual no significa necesariamente que el navegador pueda acceder a Internet, ya que puede estar conectado a una intranet sin acceso a internet. La solución a este problema es no asumir que hay acceso a internet si **`navigator.onLine`** devuelve **true**. Si devuelve false es seguro que no hay acceso a Internet, pero no al contrario.

```
// Comprobar si tenemos conexión:

if(navigator.onLine) {
    goOnline();
} else {
   goOffile();
}

// Eventos offline y online, comprueban los cambios en el estado de la red.
window.addEventListener('offline', goOffile );
window.addEventListener('online', goOnline);

//  ----------------------------------------------------------

function goOnline() {
    document.getElementById("line").src = "img/online.png";
}

function goOffile() {
    document.getElementById("line").src = "img/offline.png";
}
```

Para probar esto, podemos abrir nuestra aplicación web en Firefox y pulsar  - Más – Trabajar sin conexión.

En Chrome, podemos hacerlo con F12 – Pestaña Aplicación – Chequear la opción “Offline”.

Una forma más fiable y casera de comprobar que no tenemos conexión a Internet sería la de hacer una petición ***fetch*** a un fichero que sepamos que existe en el servidor. Esto se repetiría cada cierto tiempo usando, por ejemplo, **setInterval** en un **Web Worker**. Si no tuviéramos conexión se podría comprobar la respuesta de la petición (resp.status) y chequear el código.

No es fácil determinar que el error sea porque no hay conexión a Internet y, por lo tanto, no se puede acceder al servidor. La imagen anterior es un error especificado por Google Chrome, no es un estándar.

Existe una API que nos da información sobre la red. Está en fase experimental aún. Se trata de **`navigator.connection`**. Esta propiedad nos da algunas propiedades que nos pueden ayudar a probar si tenemos conexión o no a Internet. Comprueba sus propiedades y úsala con el método **`change:`** `navigator.connection.addEventListener('change', listener).`

6. API CacheStorage

Los navegadores actuales admiten la nueva **API Cache**. Esta API está diseñada para **almacenar pares de objetos solicitudes / respuestas (Request/Respond) HTTP que se almacenan en la memoria usada por del navegador** y es muy útil para hacer cosas como almacenar activos del sitio web sin conexión para que el sitio se pueda usar posteriormente sin una conexión de red. La caché generalmente se usa en combinación con la **API Service Worker**, aunque no necesariamente tiene que ser así.

Para empezar a utilizar la API Cache deberemos cachear si el navegador lo soporta:

```
if ('caches' in window){
    console.log("API Caché disponible")
}
else {
    console.log("API Caché NO disponible")
}
```

CacheStorage **almacena un par de objetos de solicitud y respuesta**. La Solicitud como clave y la Respuesta como valor. El objeto Solicitud se utiliza para enviar una solicitud HTTP a través de una red, la red responde con un objeto Respuesta del servidor. Ahora, los dos forman un par, muy parecido a una pregunta y una respuesta.

```
// pregunta: Por favor, Servidor necesito la imagen `image.png`
const req = new Request('/image.jpg)


// Servidor: Hola Request, he encontrado la imagen que pides. Te la envio
const res = new Response(new Blob([data]),{type:'image/jpeg'})
```

Ahora, en la Caché del cliente tendremos un par de valores:

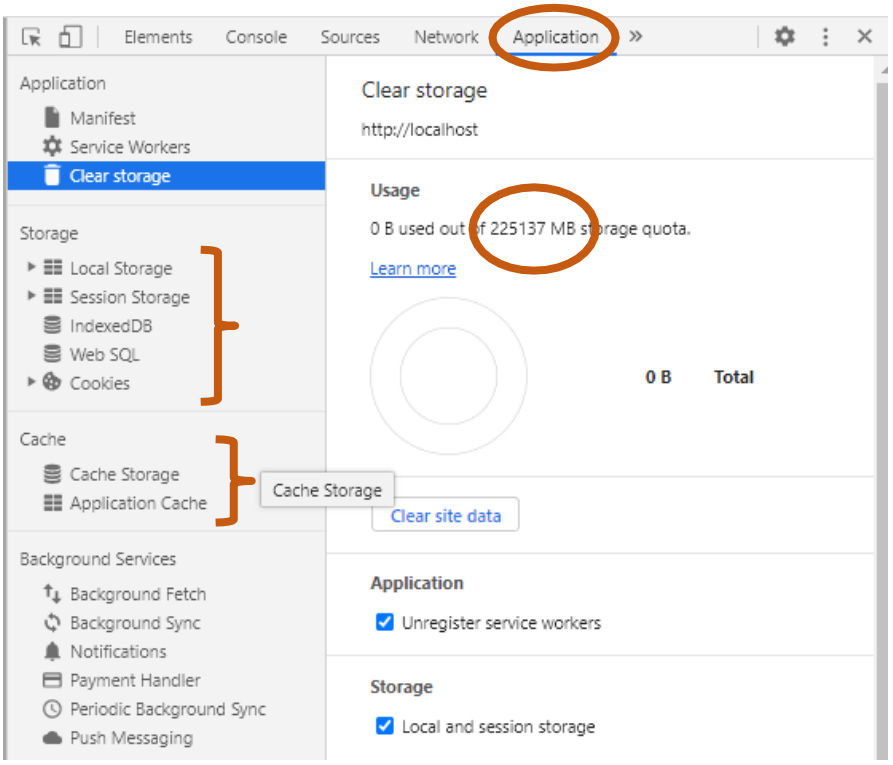
```
Cache
-------------
key | value
-------------
req | res
```

Ahora, para obtener el objeto Response ***res***, haremos referencia desde ***req***, que es la clave (key):

```
console.log(Cache[req])
```

La API de caché se creó para permitir que los Service Workers almacenen en caché las solicitudes de red para que puedan proporcionar respuestas rápidas, independientemente de la velocidad o disponibilidad de la red. Sin embargo, la API también se puede utilizar como mecanismo de almacenamiento general.

¿Cuánto se puede almacenar? En resumen, mucho, al menos un par de cientos de megabytes y potencialmente cientos de gigabytes o más. Las implementaciones del navegador varían, pero la cantidad de almacenamiento disponible generalmente se basa en la cantidad de almacenamiento disponible en el dispositivo.



Hay que tener en cuenta que en la Cache Storage **siempre almacenamos pares Request / Response** y que los objetos Response aceptan muchos tipos de datos, incluidos **Blob**, **ArrayBuffer**, FormData y cadenas.

```
const imageBlob = new Blob([data], {type: 'image/jpeg'});
const imageResponse = new Response(imageBlob);
const stringResponse = new Response('Hello world');
```

Puedes configurar el tipo MIME de la Response configurando el encabezado apropiado.

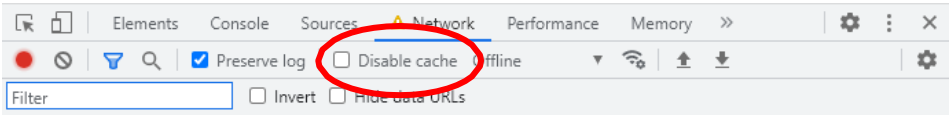
```
const options = {

   headers: {'Content-Type': 'application/json'}
}
const jsonResponse = new Response('{}', options);
```

A continuación, vamos a ver algunos métodos de la API CacheStorage. **Para ayudarte a entender mejor las operaciones que vamos a realizar puedes abrir las Herramientas para desarrolladores (DevTools) y pulsar en Application – Cache Storage.**

6.1 Objeto Caches

El objeto ***caches*** (una instancia de CacheStorage) se utiliza para acceder al CacheStorage, para recuperar, almacenar y eliminar objetos. El objeto de ***caches*** es global, está ubicado en el objeto ***window***.

Como el objetivo es recuperar los datos almacenados en la caché del navegador, es importante tener en cuanta que, si estamos OFFLINE, **debemos tener deshabilitada la opción de "Disable cache" del navegador**. Por ejemplo, para Chrome:



6.2 Crear una nueva caché

Antes de que podamos comenzar a almacenar pares de solicitud-respuesta en nuestra caché, debemos crear una instancia de caché. Cada origen puede tener varios objetos de caché dentro de su almacenamiento de caché. Podemos crear un nuevo objeto de caché usando el método *caches.open ()*:

```
const newCache = caches.open('nueva-caché');
```

El fragmento anterior recibe el nombre de la caché como parámetro único y continúa creando la caché con ese nombre. El método `caches.open()` primero comprueba si ya existe una caché con ese nombre. Si no es así, lo crea **y devuelve una Promise que se resuelve con el objeto Cache**. Después de que se ejecute el fragmento, ahora tendremos un nuevo objeto de caché al que se puede hacer referencia con el nombre ***new-cache***.

6.3 Checking a Cache

Podemos comprobar si existe un objeto Cache llamando al método **`has()`** con el nombre del objeto caché.

```
if ('caches' in window) {
    caches.has('cache_name').then((bool) => {
        // bool == true
    }).catch((err) => {
        //Error...
    })
}
```

6.4 Añadiendo elementos a la caché

Para añadir elementos a la caché usaremos los métodos: **add, addAll, has, put, match, matchAll y keys**. **Todos estos métodos devuelven una promesa**.

6.5 Cache.add()

Toma un solo parámetro que puede ser un literal de cadena de URL o un objeto ***Request***. Una llamada al método `cache.add()` hará una solicitud ***Fetch*** a la red y almacenará la respuesta en el objeto de caché asociado:

```
const myCache = caches.open('new-cache')
        .then(function(cache) {
             return cache.add(url);
        });
     })
```

También podemos hacerlo a través del objeto ***Request***:

```
const options = {
    method: "GET",
    headers: new Headers({'Content-Type': 'text/html'})
}

const myCache = caches.open('new-cache')
        .then(function(cache) {
             cache.add(new Request('/cats.json', options));
        });
     })
```

6.6 Cache.addAll()

Este método funciona de manera similar al método *`cache.add()`* excepto que toma una matriz de literales de cadena de URL de solicitud u objetos de solicitud y devuelve una promesa cuando todos los recursos se han almacenado en caché:

```
const urls = ['pets/cats.json', 'pets/dogs.json'];
newCache.addAll(urls);
```

`addAll()` sobrescribirá cualquier par clave / valor almacenado previamente en la caché que coincida con la solicitud, pero fallará si una operación **`put()`** sobrescribe una entrada de caché anterior almacenada por el mismo método **`addAll()`**. Una alternativa al uso de **`addAll()`** podría ser:

```
const myCache = caches.open('myCache')
    .then(function(cache) {
       /* Alternativa a addAll: */
       urls.map((url) => {
           console.log(url);
           return cache.add(url);
```

```
        });
})
```

6.7 Cache.put()

Permite agregar un nuevo elemento al cache. Este método funciona de manera diferente **`add()`**, ya que permite una capa adicional de control. El método *`put()`* toma dos parámetros, el primero puede ser un literal de cadena de URL o un objeto Request, el segundo es una ***Response*** ya sea de la red o generada dentro de su código. En este último caso, se almacenarán en el Cache Storage del navegador el par **Request** y **Response** pasado a **put**:

```
var req = new Request("img/image.gif");
myCache = caches.open(myCacheName)
    .then(function(cache) {
        fetch(req)
        .then(function(response) {
            if (!response.ok)
                console.error('Error fetch!');
            else
                cache.put(req, response);
        })
        .catch(function(err) {
            console.error("ERROR fetch: " + err);
        });
    })
    .catch(function(err) {
        console.error("ERROR OPEN: " + err);
    });
```

El método **`put()`** permite una capa adicional de control porque permite almacenar respuestas que no dependen de CORS u otras respuestas que dependen de un código de estado de respuesta del servidor. En definitiva, podemos almacenar una respuesta con los datos que nosotros mismos hemos generado. **Los métodos Cache.add y Cache.addAll no almacenan en caché respuestas con un Response.status que no está en el rango 200**, sin embargo Cache.put permite almacenar cualquier par Request/Response, aunque la Response tenga un estado distinto.

Cache.put devuelve una promesa que se resuelve como **undefined**.

6.8 Cache.match()

Después de agregar algunos elementos a la caché, debemos poder recuperarlos durante el tiempo de ejecución. Podemos usar el método *match ()* para recuperar nuestras respuestas en caché:

```
const file = "img/imgen.csv";

caches.open("myCache")
.then(function(cache) {
     cache.match(file)
     .then(function (resp) {
         if(resp.ok)
            console.log("Encontrado "+file+" en la caché!");
         else
            console.log("No se ha encontrado "+file+" en la caché!");
     })
     .catch(function(err) {
         console.error("ERROR: "+err);
     });
});
.catch(function(err) {
     console.error("ERROR: "+err);
});
```

En el código anterior, pasamos una variable de solicitud al método match, si la variable es una cadena de URL, se convierte en un ***Request*** y se usa como argumento. El método **match** devolverá una Promesa que se resuelve en un objeto ***Response*** si se encuentra una entrada coincidente.

El navegador utiliza diferentes factores para determinar si dos o más solicitudes coinciden. Una solicitud puede tener la misma URL que otra, pero utilizar un método HTTP diferente. El navegador considera que estas dos solicitudes son diferentes.

Cuando usamos el método **match**, también podemos pasar un objeto de opciones como segundo parámetro. Este objeto tiene pares clave-valor que le dicen a **match** que ignore factores específicos al hacer coincidir una solicitud:

```
const options = {
    ignoreVary: true, // ignora diferencias en las Cabeceras (Headers)
    ignoreMethod: true, // ignora diferencias en los métodos HTTP
    ignoreSearch: true // ignora diferencias en las consultas
}

cache.match(request, options).then(function() {…});

const request = "/img/imagen.png";

const responses = cache.matchAll(request, options);
console.log(`There are ${responses.length} matching responses.`);
```

En el caso de que coincida más de un elemento de caché, se devuelve el más antiguo. Si pretendemos recuperar todas las respuestas coincidentes, podemos usar el método **`matchAll()`**.

Nota: Tanto **match** como **matchAll** devuelven una Promise.

6.9 Cache.delete()

Podemos borrar la caché entera de la siguiente forma:

```
caches.has(myCacheName)
    .then(function() {
        caches.delete(myCacheName)
            .then(function (valor) {
                console.log("Caché eliminada con éxito!");
            })
           .catch(function() {
                console.error("No se ha podido eliminar la caché!");
            });
    })
    .catch(function(err) {
        console.error("Error al borrar la caché! " + err);
    });
```

Si ahora hacemos ***caches.has(cache-name)***, debería devolvernos falso.

NOTA: Para borrar un solo elemento de una caché, tendremos que comprobar que dicha caché existe y abrirla con el método ***open***. En el ***then*** que se nos devuelve podremos eliminar el fichero deseado almacenado en esa caché.

Nota: Cuando se elimina una caché, el método **delete ()** devuelve una Promesa con el valor ***true*** si la caché realmente se eliminó y ***false*** si algo salió mal o la caché no existe.

6.10 Cache.keys()

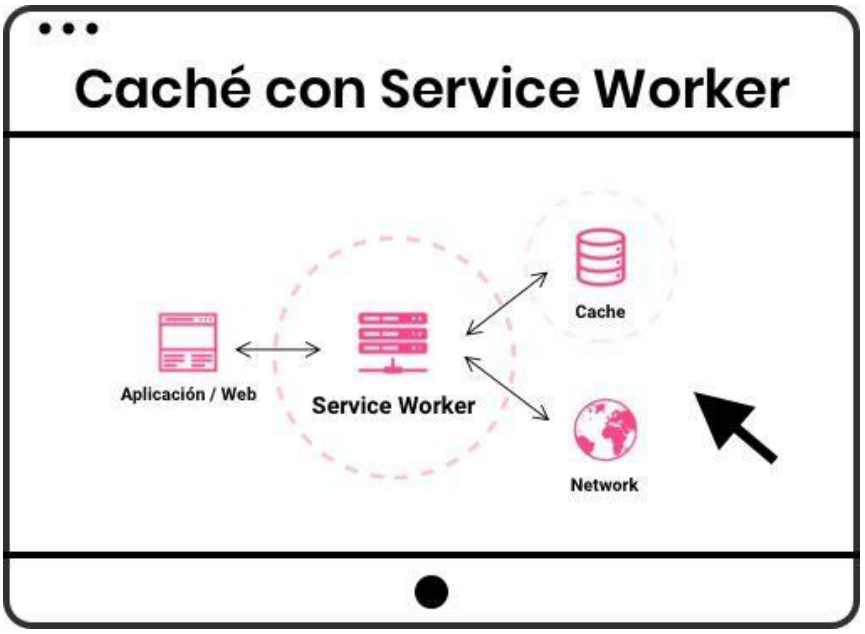
El método ***keys()*** devuelve una Promise que se resuelve en una matriz de claves con los nombres de las cachés creadas. Las solicitudes se devuelven en el mismo orden en que se insertaron.

```
caches.keys()
    .then(function (k) {
        if (k.length != 0)
          for (let i in k)
            console.log("Caché: "+ k[i]");
     })
    .catch(function(err) {
        console.error("Error al listar las cachés " + err);
    });
```

NOTA: Podemos listar el contenido (ficheros) de una memoria caché concreta. Para ello, antes habrá que abrirla y ejecutar ***keys*** sobre esa caché.

7. Service worker

Los Service workers actúan esencialmente como servidores Proxy asentados entre las aplicaciones web, el navegador y la red (*cuando está accesible*). Están destinados, entre otras cosas, a permitir la creación de **experiencias offline efectivas**, interceptando peticiones de red y realizando la acción apropiada si la conexión de red está disponible y hay disponibles contenidos actualizados en el servidor. También permitirán el **acceso a notificaciones tipo push** y APIs de sincronización en segundo plano.



7.1 Uso de Service worker

Un service worker es un **Web Worker** (*visto en temas anteriores*) manejado por eventos registrados para una fuente y una ruta. Consiste en un fichero JavaScript que controla la página web (o el sitio) con el que está asociado, **interceptando y modificando la navegación y las peticiones de recursos, y cacheando los recursos** para ofrecer un control completo sobre cómo la aplicación debe comportarse en ciertas situaciones (**la más obvia es cuando la red no está disponible**).

Un service worker se ejecuta en un contexto worker; por lo tanto, **no tiene acceso al DOM**, y **se ejecuta en un hilo distinto al JavaScript principal** de la aplicación, de manera que no es bloqueante. Está diseñado para ser completamente **asíncrono**, por lo que APIs como el **XMLHttpRequest** asíncrono y **localStorage** no se pueden usar dentro de un service worker.

Los service workers solo funcionan sobre HTTPS (*en desarrollo funcionará con localhost también***)**, por razones de seguridad. Modificar las peticiones de red en abierto permitiría ataques ***man in the middle*** realmente peligrosos. En Firefox, las APIs de service worker se ocultan y no pueden ser empleadas cuando el usuario está en modo de "*navegación en privado*".

7.2 Registro

Un service worker tiene un ciclo de vida completamente separado de tu página web.

Si quieres instalar un service worker para tu sitio, debes registrarlo antes. Esto se realiza en el lenguaje JavaScript de tu página. Cuando registres un service worker, el navegador iniciará la etapa de instalación del proceso en segundo plano.

Un service worker se registra mediante el método **`serviceWorker.register()`**. Si ha habido éxito, el service worker se descargará en el cliente e intentará la instalación/activación de las URLs accedidas por el usuario dentro de todo su origen de datos, o dentro de algún subconjunto especificado por el autor.

Para instalar un service worker, debes **registrarlo** en tu página para iniciar el proceso, **en el fichero JavaScript principal**. De esta forma, se comunica al navegador dónde reside el archivo JavaScript de tu service worker.

```
if ('serviceWorker' in navigator) {
  window.addEventListener('load', function() {
    navigator.serviceWorker.register('/worker.js').then(function(registration) {
      // Registro con éxito
      console.log('Registro del ServiceWorker con éxito: ', registration.scope);
    }, function(err) {
      // Registro fallido :(
      console.log('Registro del ServiceWorker falló: ', err);
    });
  });
}
```

Este código verifica si la API del service worker está disponible. Si está disponible, se registra el service worker `/worker.js` una vez que se carga la página.

7.3 Descarga, Instalación y Activación

En este punto, el service worker seguirá el siguiente ciclo de vida:

1. Descarga
2. Instalación
3. Activación

El service worker se **descarga** inmediatamente cuando un usuario accede por primera vez a un sitio controlado por el mismo.

La **instalación** se realiza cuando el fichero descargado es nuevo, es decir, diferente a otro service worker existente (comparado byte a byte), o si es el primero descargado para esta página/sitio.

Por lo general, durante la etapa de instalación, te convendrá almacenar en caché algunos elementos estáticos. **Si todos los archivos se almacenan correctamente en caché, se instalará el service worker**. Si no se puede descargar o almacenar en caché alguno de los archivos, el paso de instalación fallará y el service worker no se activará. Si esto ocurre, no te preocupes; se realizará un nuevo intento la próxima vez. Sin embargo, si la instalación tiene éxito, podrás estar seguro de que dichos elementos estáticos estarán en la caché.

Como parte del ejemplo más básico, debes crear una función que recoja el evento de instalación y definir los archivos que deseas almacenar en caché.

```
var CACHE_NAME = 'my-site-cache-v1';
var urlsToCache = [
  '/',
  '/styles/main.css',
  '/script/main.js'
];

self.addEventListener('install', function(event) { // También se puede usar this
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then(function(cache) {
        console.log('Opened cache');
        return cache.addAll(urlsToCache);
      })
  );
});
```

Presta atención al evento **`install;`** es habitual preparar tu service worker para usarlo cuando se dispara, por ejemplo, creando una caché que utilice la API incorporada de almacenamiento, y colocando los contenidos dentro de ella para poder usarlos con la aplicación offline.

Aquí podrás ver que se llama a `caches.open()` con el nombre de caché deseado; después se llama a `cache.addAll()` y se pasa la matriz de archivos. Esta es una cadena de promesas (`caches.open()` y `cache.addAll()`). El método `event.waitUntil()` toma una promesa y la usa para saber cuánto tarda la instalación y si se realizó correctamente.

Si todos los archivos se almacenan correctamente en caché, se instalará el service worker.

Si es la primera vez que el service worker está disponible se intenta la instalación, y tras una instalación satisfactoria se **activa**.

Si ya existe un service worker disponible, la nueva versión se instala en segundo plano, pero no se activa, en ese momento se llama *worker in waiting.* Sólo se activa cuando ya no hay más páginas cargadas que utilicen el antiguo service worker. En cuanto no hay más páginas de este estilo cargadas, el nuevo service worker se activa (pasando a ser el *active worker).* La activación del service worker se puede forzar antes mediante **`ServiceWorkerGlobalScope.skipWaiting()`**. Con este método haremos que el Service Worker que está en espera de ponerse activo pase inmediatamente a ser el Service Worker activo. Las páginas existentes se pueden llamar por el worker activo usando **`Clients.claim()`**, con esto se garantiza que las actualizaciones del SW entren en vigor inmediatamente para todos los clientes activos.

También hay un evento **`activate`** que se disparará cuando el service worker se activa. El momento en el que este evento se activa es, en general, un buen momento para limpiar viejas cachés y demás cosas asociadas con la versión previa de tu service worker.

Tu service worker puede **responder** a las peticiones usando el evento **`fetch`**. Puedes modificar la respuesta a estas peticiones de la manera que quieras, usando el método **`FetchEvent.respondWith`**. De esta forma, cada vez que se solicite un nuevo recurso del servidor, antes se puede comprobar si está ya almacenado en caché, en caso contrario, realiza la petición y la guarda en caché.

7.4 Solicitudes de devolución y caché

Ahora que has instalado un service worker, probablemente desees mostrar una de tus respuestas almacenadas en caché.

Cuando se instala un service worker y el usuario actualiza la página o se dirige a una diferente, el service worker comienza a recibir eventos **fetch** (a continuación, se muestra un ejemplo):

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.match(event.request)
      .then(function(response) {
       /* caches.match () siempre se resuelve, pero en
        caso de éxito la respuesta tendrá valor */
       if (response) {
         return response;
       } else {
           return fetch(event.request).then(function (response) {
           /* La respuesta puede usarse solo una vez, necesitamos
             guardar el clon para poner una copia en la caché */
               let responseClone = response.clone();

               caches.open(CACHE_NAME).then(function (cache) {
                   cache.put(event.request, responseClone);
```

```
                });
                return response;
            }).catch(function (err) {
                console.error('ERROR: ' + err);
            });
        }
    }));
});
```

Aquí hemos definido nuestro evento fetch y en event.respondWith(), pasamos una promesa de caches.match(). Este método examina la solicitud y encuentra cualquier resultado almacenado en caché de cualquiera de los creados por tu service worker.

Si existe una respuesta, se devuelve el valor almacenado en caché. Si no existe, se devuelve el resultado de una llamada a fetch, que realizará una solicitud de red y devolverá los datos si se puede recuperar algo de la red. Este es un ejemplo simple y en él se usa cualquier recurso que hayamos almacenado en caché durante la instalación.

Si deseamos almacenar en caché solicitudes nuevas de forma acumulativa, podemos hacerlo administrando la respuesta de la solicitud de fetch y luego agregándola a la caché, como se muestra a continuación:

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.match(event.request)
      .then(function(response) {
        if (response) {
          return response;
        }

        // IMPORTANTE: Clona la petición.
        var fetchRequest = event.request.clone();

        return fetch(fetchRequest).then(
          function(response) {
            // Chequeamos si recibimos una respuesta válida.
            if(!response || response.status !== 200 || response.type !== 'basic') {
              return response;
            }

            // IMPORTANTE: Clonamos la respuesta.
            var responseToCache = response.clone();

            caches.open(CACHE_NAME)
              .then(function(cache) {
                cache.put(event.request, responseToCache); // Actualizamos la cache.
              });

            return response;
          }
```

```
        );
      })
    );
});
```

Esto es lo que estamos haciendo:

1. Agregamos una devolución de llamada **`a.then()`** en la solicitud **fetch**.
2. Cuando recibimos una respuesta, realizamos las siguientes verificaciones:
   a. Nos aseguramos de que la respuesta sea válida.
   b. Verificamos que el estado sea **200** en la respuesta.
   c. Nos aseguramos de que el tipo de respuesta sea *basic*, lo que indica que es una solicitud proveniente de nuestro origen. Esto también significa que las solicitudes a recursos de terceros no se almacenan en caché.
3. Si pasamos las verificaciones, clonamos la respuesta. Esto es así porque, al ser la respuesta una transmisión, el cuerpo solo se puede consumir una vez. Debido a que deseamos devolver la respuesta para que el navegador la use, además de pasarla a la caché para su aplicación, debemos clonarla a fin de enviar una al navegador y otra a la caché.

Los service workers también pueden usarse para cosas como:

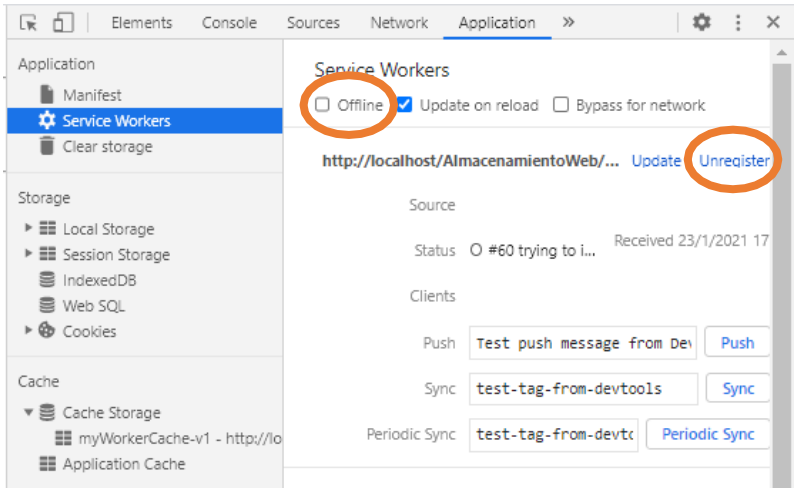
- Sincronización de datos en background.
- Responder a peticiones de recursos desde otros orígenes.
- Recibir actualizaciones centralizadas de datos costosos de calcular tales como geolocalización o giroscopio, de manera que muchas páginas puedan hacer uso de un mismo conjunto de datos.
- Plantillas HTML/CSS personalizadas basadas en ciertos patrones URL.
- Mejoras de rendimiento, por ejemplo, pre-fetching de recursos que es probable que el usuario requiera en un futuro próximo, como las próximas imágenes de un álbum de fotos.

7.5 Eliminar un ServiceWorker

Para eliminar un ServiceWorker podemos usar el siguiente código:

```
navigator.serviceWorker.getRegistrations().then(function(registrations) {
    for(let registration of registrations) {
         registration.unregister()
    }
})
```

También podemos irnos a las herramientas de desarrollador del navegador (DevTools) – Pestaña *Aplications* – Service Workers – Pulsar la opción ***Unregister***.



NOTA: Si marcamos la opción "Offline" se simulará una pérdida de conexión a la red. Así podremos testear cómo se comporta nuestra aplicación al pulsar **F5 o Ctrl-F5**.

Cuidado!: De forma predeterminada, la ruta actual del Service Worker es el directorio en el que se encuentra. Por ejemplo, si accedes a tu script de Service Worker desde **https://your.domain.com/static/service-worker.js**, su alcance (**scope**) predeterminado será **/static**. Por lo tanto, si escribes **`cache.add('index.html')`**, en realidad solicitará **https://your.domain.com/static/index.html**, lo que dará como resultado un error si intentaba obtener **https://tu.dominio.com/index.html**. **Por este motivo, se aconseja poner el archivo .js que ejecuta el Service Worker en la raíz del servidor, así su alcance será cualquier directorio y podrá guardar cualquier fichero en la caché que genera**. En todo caso, si se quiere cambiar el alcance del Service Worker, se puede configurar la cabecera ***`Service-Worker-Allowed`***.

Existen otros sistemas de almacenamiento en el cliente web que permiten almacenar datos más complejos como, por ejemplo, bases de datos locales. Para profundizar más en este tema, te aconsejo que investigues sobre estos temas:

- **IndexedDB**
- **Web SQL**

8. Notificaciones

La API de Notificaciones permite a una página web enviar notificaciones que se mostrarán fuera de la web al nivel del sistema. Esto permite a las aplicaciones web enviar información al usuario, aunque éstas estén de fondo.

Esta característica está disponible en Web Workers.

El sistema de notificaciones variará según el navegador y la plataforma en la que estemos, pero esto no va a suponer ningún problema. La API de notificaciones está escrita de manera que sea compatible con la gran mayoría de sistemas.

8.1 Pidiendo permiso

Antes de que una app pueda lanzar una notificación, el usuario tiene que darle permiso para ello. Esto es un requisito común cuando una API intenta interactuar con algo fuera de una página web — al menos una vez, el usuario tendrá que permitir a la aplicación mostrar notificaciones, de esta forma, el usuario decide qué aplicaciones le pueden mostrar notificaciones y cuáles no.

Puedes comprobar si ya tienes permiso comprobando la propiedad **`Notification.permission`**. Esta puede tener uno de los siguientes valores:

- **default:** No se le ha pedido permiso al usuario aún, por lo que la app no tiene permisos.
- **granted:** El usuario ha permitido las notificaciones de la app.
- **denied:** El usuario ha denegado las notificaciones de la app.

Si la aplicación aún no tiene permiso para mostrar notificaciones, tendremos que hacer uso de **Notification.requestPermission()** para pedir permiso al usuario.

```
if (!("Notification" in window)) {
    alert("Este navegador no soporta las notificaciones del sistema");
}
else {
    Notification.requestPermission().then(function(result) {
        console.log(result);
    });
}
```

8.2 Crear una nueva notificación

Crear una notificación es fácil, simplemente usa el constructor **`Notification`**. Este constructor espera un título que mostrar en la notificación y otras opciones para mejorar la notificación, como un **icon** o un texto **body**.

```
var options = {
    body: theBody, // Mensaje.
    icon: theIcon // Icono de la notificación (opcional).
}

var n = new Notification("Título de la notificación",options);

setTimeout(n.close.bind(n), 5000); // Cierra la notificación pasados 5 segundos.
```

Firefox y Safari cierran las notificaciones automáticamente tras cierto tiempo (unos 4 segundos). Esto también puede suceder a nivel de sistema operativo (en Windows duran 7 segundos por defecto). En cambio, en algunos navegadores, puede ser que se muestren de forma permanente. Para asegurarnos de que las notificaciones se cierran en todos los navegadores, al final de las funciones de arriba, llamamos a la función **`notification.close`** dentro de **`setTimeout()`** para cerrar la notificación tras 5 segundos. Date cuenta también del uso que hacemos de **`bind()`** para asegurarnos de que la función **`close()`** está asociada a la notificación, ya que bind() convierte en ***this*** al primer parámetro que se le pasa.

8.3 Eventos de Notificación

Las especificaciones de la API de notificaciones listan dos eventos que pueden ser lanzados en la instancia **`Notification`**:

- **click:** Lanzado cuando el usuario hace click en la notificación.
- **error:** Lanzado cuando algo falla en la notificación; habitualmente es porque la notificación no se ha podido mostrar por algún motivo.
- **show:** Se lanza cuando se empieza a mostrar la notificación.
- **close:** Se lanza cuando se cierra la notificación.

8.4 Propiedades de Notification

- **`Notification.actions`**: Arreglo de acciones de la notificación, como se especifica en el parámetro de opciones del constructor.
- **`Notification.badge`**: URL de la imagen usada para representar la notificación cuando no hay espacio suficiente para mostrarla.
- **`Notification.title`**: El título de la notificación como está especificado en el parámetro options del constructor.
- **`Notification.dir`**: La dirección del texto de la notificación como está especificado en el parámetro options del constructor.
- **`Notification.lang`**: El código del lenguaje de la notificación como está especificado en el parámetro options del constructor.
- **`Notification.body`**: The body string de la notificación como está especificado en el parámetro options del constructor.
- **`Notification.tag`**: El ID de la notificación (si hay) como está especificado en el parámetro options del constructor.
- **`Notification.icon`**: La URL de la imagen usada como ícono de la notificación como está especificado en el parámetro options del constructor.

- **`Notification.image`**: URL de una imagen para mostrar como parte de la notificación, al igual que se especifica en el parámetro de opciones del constructor.
- **`Notification.data`**: Retorna un clon estructurado de los datos de la notificación.
- **`Notification.requireInteraction`**: Un **`booleano`** indicando en dispositivos pantallas lo suficientemente grandes, una notificación debería permanecer activa hasta que el usuario haga click o la descarte.
- **`Notification.silent`**: Especifica si la notificación debería ser silenciada, por ejemplo, sin generar sonidos o vibraciones, independientemente de la configuración del dispositivo.
- **`Notification.timestamp`**: Especifica el tiempo en la cual una notificación fue creada o aplicable (pasado, presente o futuro).

En algunos navegadores como Chrome para Android, al usar notificaciones se puede lanzar un error **`TypeError`** cuando se llama al constructor de Notification. Las notificaciones solamente se pueden crear desde un service worker o web worker.

Las siguientes propiedades están listadas en las especificaciones más actualizadas, pero aún no están soportadas por algunos navegadores. Es aconsejable verificarlas regularmente para ver si el estado de ellas ha sido actualizado:

- **`Notification.noscreen`**: Especifica si la activación de la notificación debe habilitar la pantalla del dispositivo o no.
- **`Notification.renotify`**: Especifica si se debe notificar al usuario después de que una notificación nueva reemplace a una anterior.
- **`Notification.sound`**: Especifica un recurso de sonido para reproducir cuando se activa la notificación, en lugar del sonido de notificación del sistema predeterminado.
- **`Notification.sticky`**: Especifica si la notificación debe ser 'fija', es decir, no fácilmente eliminable por el usuario.
- **`Notification.vibrate`**: Especifica un patrón de vibración para los dispositivos con hardware de vibraciones para emitirlo.

NOTA: Todas estas propiedades son de solo lectura.

8.5 Reemplazando notificaciones existentes

Normalmente los usuarios no quieren recibir muchas notificaciones en poco tiempo — por ejemplo, una aplicación de mensajería que te notifica cada mensaje que te llegue, y te llegan un montón. Para evitar el spam de notificaciones, se puede modificar la cola de notificaciones, reemplazando una o varias notificaciones pendientes, por una nueva notificación.

Para hacer esto, se puede añadir una etiqueta a cualquier nueva notificación. Si ya hay una notificación con la misma etiqueta y aún no se ha mostrado, la nueva reemplazará a la anterior. Si la notificación con la misma etiqueta ya ha sido mostrada, se cerrará la anterior y se mostrará la nueva.

```
var i = 0;

var interval = window.setInterval(function () {
    // Gracias a la etiqueta, deberíamos de ver sólo la notificación "Holiws! 9"
    var n = new Notification("Holiwis! " + i, {tag: 'soManyNotification'});
    if (i++ == 9) {
      window.clearInterval(interval);
    }
  }, 200);
```

NOTA: Podemos hacer click en una notificación y capturar ese evento. Esto funciona por ahora, pero en la especificación ha desaparecido y pronto estará obsoleto. Un ejemplo de cómo hacer click en una notificación:

```
function showNotification() {
    const notification = new Notification("New message incoming", {
       body: "Hi there. How are you doing?"
    })
    notification.onclick = (e) => {
       window.location.href = "https://google.com";
    };
}
```

Para saber más sobre el tema de las notificaciones te recomiendo que investigues sobre la nueva **API Web Push**, la cual se basa en el uso de Service Workers y te permitirá enviar notificaciones entre cliente y servidor.

Fuentes:

https://developer.mozilla.org/

https://developers.google.com/