UD11. USO DE APIS DE JAVASCRIPT

1. ¿QUÉ SON LAS APIS?

API(Application Programming Interface) es el conjunto de funciones, métodos, objetos y componentes que permiten manejar un determinado software o componente. Así, el método fetch, los objetos Response, Request y Headers, con sus propiedades y métodos, forman junto a otros elementos, la API conocida como **Fetch** que facilita la gestión de peticiones y respuesta de tipo AJAX(Asynchronus JavaScript and XML).

Incluso APIs mucho más extensas, como es el caso de la **API DOM** de JavaScript que permite manipular los elementos de las aplicaciones web y que está formada por cientos de métodos, propiedades y objetos.

Para crear aplicaciones web del lado del cliente es fundamental conocer el lenguaje JavaScript al máximo nivel posible, además hay que comprender y manipular el DOM perfectamente así como conocer cómo funciona AJAX para poder crear aplicaciones que utilizan datos y contenidos de servidores de Internet. A partir de aquí, JavaScript proporciona numerosas APIs más para realizar otro tipo de labores. Pero dependerá de nuestros intereses utilizar unas u otras. La idea es entender cómo funcionan las APIs y cómo se aplican las más importantes, para así, si tenemos que aprender otras, no nos sea complicado.

2. APIS DE HTML5

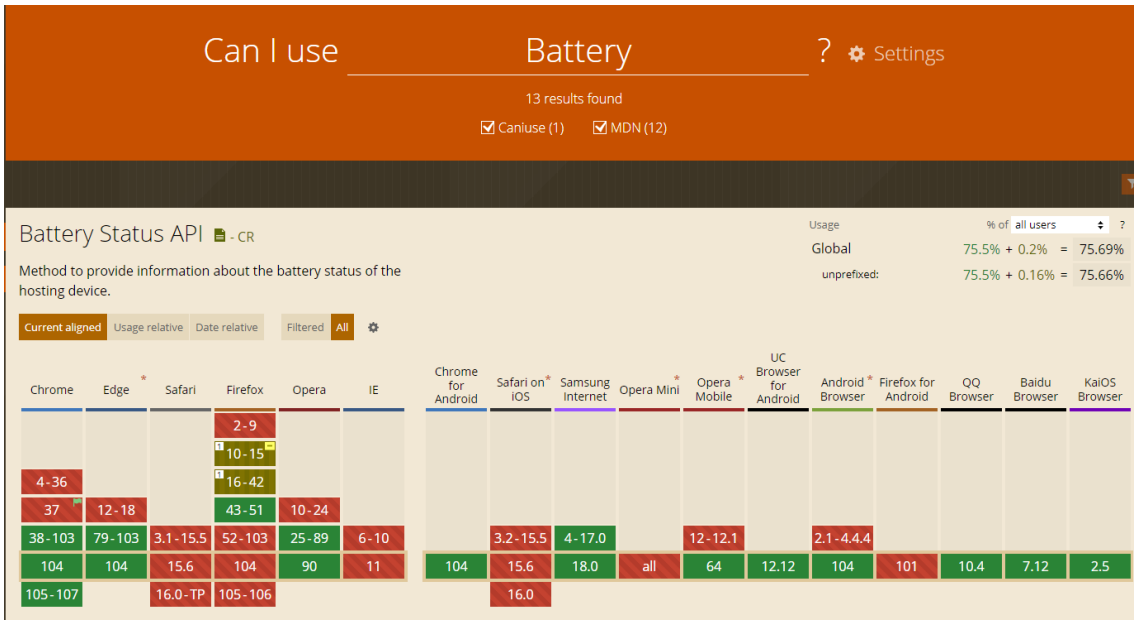
El estándar HTML5 cubre el lenguaje HTML, CSS y el propio JavaScript. A diferncia de la entidad ECMA que se encarga de estandarizar el lenguaje JavaScript en su norma ECMAScript, las entidades normalizadoras HTML5 (fundamentalmente W3C y WHATWG) definen APIs creadas para JavaScript a fin de manipular aspectos de las aplicaciones web.

Ya hemos trabajado con APIs HTML5: la API Fetch y el propio DOM, que proporciona objetos (**window**, **document**, etc) y los métodos y propiedades para trabajar con ellos y que nos permiten manipular los elementos de un documento HTML y gestionar los eventos.

Ejemplos de ellas son:

- **Web Storage.** Permite el almacenamiento de datos en el navegador. Su finalidad es no depender de las cookies para realizar esta labor.
- **Web Workers.** API para la creación de tareas pesadas que se ejecutan en segundo plano y que, en primer plano, ralentizarían el trabajo.
- **Geolocation.** API que permite gestionar coordenadas de localización (longitud y latitud) con el fin de utilizarlas en aplicaciones web.
- **Web Sockets.** Permite la creación de sockets de red para la comunicación direta entre el cliente y el servidor.
- **API Canvas.** Para la manipulación del elemento canvas de HTML5 que permite la creación de gráficos en 2 dimensiones.

- **Web GL.** API que incorpora las capacidades de Open GL para crear gráficos y animaciones en 3 dimensiones en los elementos canvas.
- **FileSystem API.** APi para poder utilizar desde JavaScript archivos locales
- Application Cache. Elemento fundamental para la creación de aplicaciones web que permitan el trabajo offline.
- **Media API.** Permite manipular elementos de vídeo y audio en una aplicación web.
- **Text Track API.** Permite manipular subtítulos para componentes de vídeo y audio.
- **Drag and Drop.** Para facilitar la programación de componentes arrastrables.
- **FullScreen.** API para poder manipular las aplicaciones web en estado de "pantalla completa".
- **Battery Status.** API para poder obtener información de la batería del dispositivo.



Hay muchas más. No todas están soportadas por todos los navegadores. La página https://caniuse.com/ nos permite saber la compatibilidad con los navegadores de las principales APIs de HTML5.

Año a año, se van añadiendo nuevas APIs que proporcionan nuevas funcionalidades y, también, se modifican las APIs existentes para adaptarlas a las nuevas capacidades del lenguaje JavaScript o para modernizar alguno de sus aspectos. Conviene documetarse de vez en cuando sobre estos cambios.

3. API PARA CANVAS

3.1. ¿QUÉ ES CANVAS?

Canvas es un elemento de HTML5 que permite crear gráficos y animaciones utilizando comandos de JavaScript. El elemento canvas HTML es un marco que se posiciona en el sitio que deseamos y que podemos modificar de tamaño:

```
<canvas id="lienzo" width="500" height="300"></canvas>
```

Desde JavaScript podemos acceder a este componente y, a través de los métodos que nos proporciona la API de Canvas, realizar creaciones gráficas en el espacio que proporciona el elemento **canvas**.

Este elemento tuvo un éxito muy grande desde su aparición en el estándar. Actualmente se sigue usando mucho, pero teien bastante competencia con el éxito que también están teniendo los gráficos **SVG**.

3.2. EMPEZAR A TRABAJAR CON CANVAS

En primer lugar, debemos obtener una referencia al elemento canvas desde JavaScript:

```
const lienzo=document.getElementById("lienzo");
```

Después debemos obtener el contexto del elemento.

```
const ctx=lienzo.getContext("2d");
```

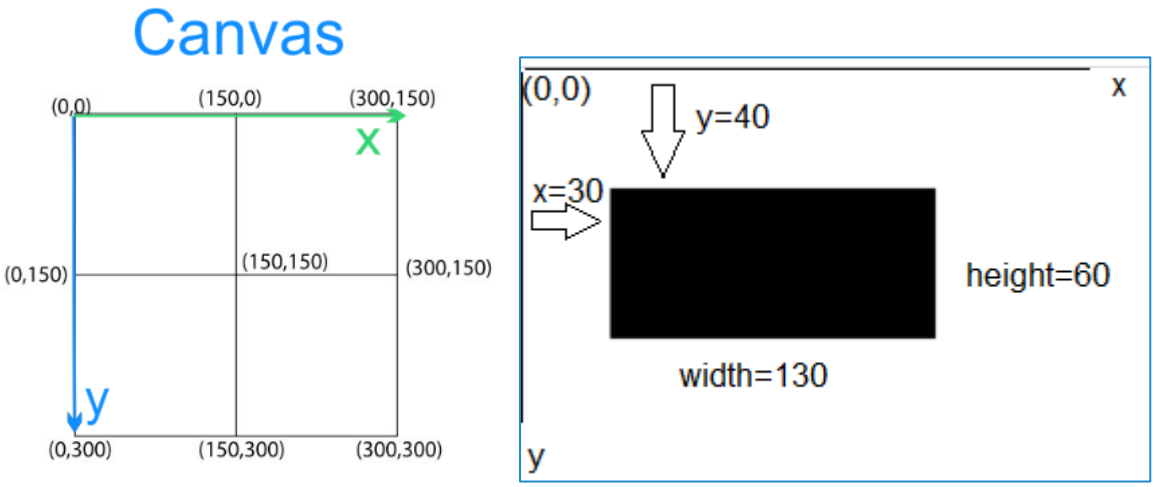
El contexto es el objeto que permite dibujar en el lienzo. Proporciona todos los métodos y propiedades de dibujo. El método **getContext**, presente en los elementos de tipo canvas, es el encargado de obtener el contexto. A este método se le pasa un string que puede contener los valores **2d** o **3d**. Normalmente se usa el valor 2d porque es el que usa para gráficos en 2 dimensiones, que son los habituales. El valor 3d permite el dibujo de gráficos 3d con ayuda de la librerái **WebGL**.

A partir de este momento, se pueden utilizar los métodos que permiten elegir las propiedades de trazo y relleno de los objetos que dibujemos y los métodos que permiten dibujar dichos objetos.

En muchas ocasiones tenemos que indicar coordenadas y, por ello, hay que entender cómo funciona el modelo de coordenadas de canvas.

El origen de coordenadas está en la esquina superior izquierda. Se llama eje **x** al horizontal e **y** al vertical. Cuando se dan posiciones de puntos, se indica siempre primero la coordenada **x** y luego la **y**. La dirección de las coordenadas positivas va hacia abajo (eje **y**) y a la derecha (eje **x**).

El tamaño en píxeles de las coordenadas del **canvas** lo ajustan las propiedades width y height de la etiqueta canvas de HTML.



ctx.fillRect(30,40,130,60)

En la imagen anterior se muestra cómo se coloca un rectángulo en un área **canvas** (señalado en gris claro) mediante el método **fillRect**. Los 2 primeros valores indican la posición de la esquina superior izquierda del mismo (coordenadas 30,40) el siguiente en anchura (130) y el último, la altura (60).

3.3. DIBUJO DE FORMAS

3.3.1. PROPIEDADES DE LAS FORMAS

Una forma es una figura geométrica pintada sobre el lienzo. Las formas tienen un color de fondo o relleno y un borde. El relleno se indica con un color y el borde tiene color y también grosor. Para indicar el color de relleno actual se usa la propiedad **fillStyle.** A esta propiedad se le puede indicar un color al estilo de CSS, por ejemplo "**blue**" o "**#0000FF**". La propiedad **strokeStyle** es similar y sirve para indicar el color del borde. Hay otra propiedad llamada **lineWidth** que permite indicar la anchura del borde.

Una vez indicadas estas 3 propiedades, las figuras dibujadas utilizarán esos colores y grosores de línea. Si no los especificamos, se utilizarán los valores por defecto configurados en el navegador del usuario.

3.3.2. RECTÁNGULOS

Canvas solo dispone de una forma básica, el rectángulo.Se dispone de 2 métodos para dibujar rectángulos:

- **fillRect**. Dibuja un rectángulo sólido. Para hacerlo se indica la coordenada superior izquierda del rectángulo y, la anchura y la altura del mismo.
- **strokeRect**. Dibuja un rectángulo transparente y pinta el borde.

Ejemplo:

```
<body>
    <canvas id="lienzo" width="500" height="300"></canvas>
</body>
<script>
        const lienzo=document.getElementById("lienzo");
        const contexto=lienzo.getContext("2d");
        contexto.fillStyle="red";
        contexto.lineWidth="2";
        contexto.strokeStyle="gray";
        contexto.fillRect(10,10,150,100);
        contexto.strokeRect(30,30,150,100);
</script>
```

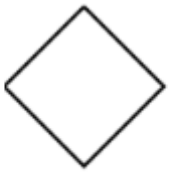
3.3.3. TRAZOS

Un trazo es una serie de dibujos de líneas que forman una figura común. Los trazos deben comenzar invocando el método **beginPath**. Este método sirve para marcar dónde comienza el trazo. Desde ese momento todos los comandos se entienden que van dirigidos a formar el trazo.

El trazo se finaliza invocando, sin argumentos, al método **closePath**, el cual cierra el trazado. Tras esa instrucción las siguientes acciones se entiende que ya no van dirigidas a ese trazo. El trazo se dibuja realmente invocando al método stroke (dibuja el contorno) o a fill (pinta el relleno). Tanto **stroke** como **fill** usan la configuración de relleno y trazo actual establecida por los métodos **strokeStyle**, **fillStyle** y **lineWidth** ya comentados.

Tras la instrucción **beginPath**, los métodos **moveTo** y **LineTo** permiten dibujar trazos poligonales. El primero (**moveTo**) permite colocar el cursor en una coordenada concreta. El segundo (**lineTo**) marcará una línea dese la última posición del cursor a la posición indicada por los parámetros de lineTo. Los parámetros de ambos métodos son una coordenada "**x**" y otra "**y**".

Ejemplo:



Hay un método que dibuja trazos rectangulares. Se llama **rect**. Los parámetros son los habituales en el relleno de rectángulos: coordenadas de la esquina superior izquierda (**x** e **y**), anchura y altura.

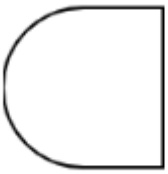
3.3.4. ARCOS

El método **arc** también se usa para crear trazos. De hecho, se suele combinar con los anteriores. Tiene como parámetros: las coordenadas (**x** e **y**) en las que se debe situar el centro del arco, el tamaño del radio, el ángulo inicial y el ángulo final. Los ángulos se miden en radianes, una mediad muy matemática que se usa mucho en ciencias matemáticas y en programación. Cabe decir que, en esa unidad, 2 veces PI equivale a 360 grados. 90 grados serían PI/2. Hay un último parámetro que indica, con un valor booleano, la dirección del arco: el valor **true indica que se ha de dibujar en la misma dirección que las agujas del reloj**.

Ejemplo:

```
<canvas id="lienzo" width="500" height="300"></canvas>
</body>
<script>
        const lienzo=document.getElementById("lienzo");
        const contexto=lienzo.getContext("2d");
        contexto.beginPath();
        contexto.lineWidth="2";
        contexto.strokeStyle="blue";
        contexto.moveTo(50,10);
        contexto.lineTo(100,10);
        contexto.lineTo(100,110);
        contexto.lineTo(50,110);
        contexto.stroke();
```

```
        contexto.closePath();
        contexto.beginPath();
        contexto.arc(50,60,50,Math.PI*3/2,Math.PI/2,true)
        contexto.stroke();
        contexto.closePath();
</script>
```

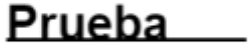


3.4. DIBUJO DE TEXTO

La propiedad font permite indicar las propiedades del texto que vayamos a dibuar en el canvas. Las propiedades a indicar se separan con espacios (al estilo de la propiedad font de CSS) dentro de un string.

El método fillText dibuja y rellena un texto cuyo contenido es el primer parámetro. Después se indican 2 coordenadas que serán la posición inferior izquierda para la línea base del texto.

```
<canvas id="lienzo" width="500" height="300"></canvas>
</body>
<script>
        const lienzo=document.getElementById("lienzo");
        const contexto=lienzo.getContext("2d");
        contexto.beginPath();
        contexto.lineWidth="2";
        contexto.strokeStyle="black";
        contexto.moveTo(50,50);
        contexto.lineTo(200,50);
        contexto.stroke();
        contexto.strokeStyle="red";
        contexto.font="bold 30px sans-serif";
        contexto.fillText("Prueba",50,50);
        contexto.closePath();
</script>
```



3.5. DIBUJAR IMÁGENES

Una de las opciones más potentes de canvas es poder colocar una imagen (gif, jpg, etc.) en el lienzo. En este sentido disponemos de un constructor de objeto llamado image que crea un objeto de imagen equivalente al elemento img de HTML. Podemos modificar las propiedades habituales de las imágenes, por ejemplo, la propiedad src.

Canvas proporciona un método llamado **drawImage** que, como primer parámetro, recibe un objeto de imagen y después 2 coordenadas que marcarán dónde quedará la esquina superior izquierda de la imagen. Además, admite 2 parámetros más con los que podemos modificar la anchura y altura de la imagen.

```
<canvas id="lienzo" width="500" height="300"></canvas>
</body>
<script>
        const lienzo=document.getElementById("lienzo");
        const contexto=lienzo.getContext("2d");
        const img=new Image();
        contexto.beginPath();
        img.src="https://img.icons8.com/material-outlined/96/000000/dog.png";
        img.addEventListener("load",(ev)=>{
                contexto.lineWidth=2;
                contexto.strokeStyle="black";
                //cuadrado contenedor
                contexto.strokeRect(200,150,96,96);
                //dibujo de la imagen
                contexto.drawImage(img,200,150);
        })
</script>
```

En el código, las instrucciones de dibujo se colocan en la función callbak del evento load de la imagen porque hay que asegurar que el código se ejecuta tras la carga de dicha imagen.

3.6. BORRAR

El método clearRect dibuja un área rectangular sobre el canvas, con la intención de eliminar todo lo que hay dentro. Es decir, borra esa área y la deja con color transparente. Los parámetros son los habituales para indicar rectángulos:

```
<canvas id="lienzo" width="500" height="300"></canvas>
</body>
<script>
        const lienzo=document.getElementById("lienzo");
        const contexto=lienzo.getContext("2d");
        contexto.fillStyle="red";
        contexto.lineWidth=2;
        contexto.strokeStyle="gray";
        contexto.rect(0,0,200,200);
        contexto.fill();
```

```
        contexto.stroke();
        contexto.clearRect(100,100,200,200);
</script>
```

3.7. CONCLUSIONES SOBRE CANVAS

Hay muchas más funciones y posibilidades para crear gráficos en Canvas. Evidentemente, crear un gráfico complicado usando estos métodos es difícil, pero combinado con otras utilidades de JavaScritp (temporizadores, eventos,etc) con formas muy sencillas se logran efectos muy llamativos e interesantes.

Hay programas gráficos profesionales que permiten dibujar en un entorno de trabajo más amigable y después convertir el resultado a formato canvas. Esto es, por ejemplo, lo que permite el software **Adobe Animate and Mobile Device Packaging** (antiguo **Adobe Flash**), se crean escenarios utilizando las potentes herramientas de este software y luego se le pide que cree un archivo con las instrucciones JavaScript que lo conseguirían dibujar en un elemento canvas.

Otra cuestión es que exiten APIs de terceros para trabajar con gráficos avanzados en las aplicaciones mucho más avanzadas y fáciles que Canvas (**Tween**, **paper.js**, **Raphael.js**, etc). Aunque, como siempre ocurre al usar librerías externas, siempre es más veloz para el navegador usar las APIs nativas.

4. WEB STORAGE

4.1 INTRODUCCIÓN A WEB STORAGE

Se trata de una API con una finalidad similar a la de las cookies, pero con una forma de trabajar distinta. Al igual que las cookies, sirve para almacenar claves nombre/Valor, pero con menos restricciones y más ventajas que las cookies.

Los datos que se almacenan, no se envían al servidor en ningún momento, se qeudan en la máquina del cliente. No forman parte de ninguna petición ni respuesta http (como sí ocurre con las cookies). Además, pueden almacenar datos de incluso varios GB, superando con creces la limitación de 4 KB de las cookies.

En definitiva, es un mecanismo de almacenamiento de información persistente que graba información en el ordenador del usuario, pero cuyos datos no son accesibles en caso de escucha de la comunicación entre cliente y el servidor.

Para manejar esta API, el objeto window aporta estas propiedades:

- **LocalStorage**. Mediante este objeto, los datos se almacenan de forma permanente
- **SessionStorage**. Los datos están disponibles hasta que se cierra la sesión, lo cual ocurre normalmente, cuando se cierra el navegador.

El funcionamiento de ambos objetos es idéntico, solo cambia la duración de los datos.

4.2. GRABAR Y LEER DATOS

El manejo de esta API es muy sencillo. Para añadir o modificar un dato se usa el método setItem que recibe como primer parámetro el nombre o clave que damos al datos y como segundo, el valor:

```
localStorage.setItem("nombre","Gema");
```
Para recoger un dato almacenado:

```
localStorage.getItem("nombre");
```
O bien:

```
LocalStorage.nombre;
```
Se admite grabar valores de texto. Pero el resto de datos simples también funcionan:

```
localStorage.setItem("casado", true);
localStorage.setItem("salario",2200);
```
Eso sí, la clave siempre es de tipo texto.

Podemos almacenar datos más complejos gracias a la ayuda del objeto JSON. Si tenemos un objeto llamado **datosPersonales** cuyos datos queremos almacenar, valdría:

localStorage.setItem("datos", **JSON.stringify**(datosPersonales));

Se usa el método **stringify** para convertir un objeto en su equivalente **JSON**. El proceso contrario sería:

```
const datosPersonales=JSON.parse(localStorage.getItem("datos"));
```
Un detalle importante a tener en cuenta es que solo páginas lanzadas desde un servidor web pueden grabar este tipo de datos. Por lo que, incluso para practicar, debemos alojar y probar nuestro código en un servidor web. Puede valer el plugin **Live Server de Visual Studio Code.**

4.3. BORRAR ITEMS

El borrado de claves almacenadas se realiza con removeItem:

```
localStorage.removeItem("nombre");
```
Como a los ítems podemos acceder directamente como propiedades del objeto localStorage, podemos borrar el ítem con el operador delete.

```
delete localStorage.nombre;
```
Podemos incluso borrar todas las entradas de golpe mediante el método clear:

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

4.4. CONCLUSIÓN SOBRE WEB STORAGE

Como vemos, el manejo de esta API es muy sencillo y, además, abre enormes posibilidades de trabajo que habitualmente se hacían con cookies y que en muchos casos podemos olvidar.

No siempre podemos olvidar las cookies, porque es habitual que en el servidor se requiera de las cookies para algunas labores. Pero, en el caso de datos que guardan detalles de la acción del usuario y no requieren la comunicación de estos datos en el servidor, indudablemente ofrecen más ventajas los objetos **localStorage** y **sessionStorage** que las cookies.

Aunque todos los ejemplos de este apartado se han hecho con **localStorage**, el código funciona igual para **sessionStorage**, simplemente reemplazando el nombre de la propiedad. La diferencia sería que los datos de **sessionStorage** se borran al cerrar la sesión.

5. GEOLOCATION

La Geolocalización es una capacidad de detectar la posición geográfica de una persona o un dispositivo. Es útil en las aplcaciones para aplicar utilidades que pueden, por ejemplo, ayudar a detectar elementos cercanos al usuario que le otorguen determinados servicios. Por ejemplo, podemos crear una aplicación de búsqueda de gasolineras cercanas a la posición del usuario. Otro ejemplo es el de algunas aplicaciones web qu permiten escuchar la radio y, mediante la geolocalización, permiten escuchar la emisora más cercana.

Detectar la posición del usuario entra en el terreno de la privacidad de dicho usuario, por lo que requiere que el propio usuario permita el uso de su localización a la aplicación web.

El DOM proporciona un objeto llamado **geolocation** como propiedad del objeto **navigator**.

5.1. OBTENER LA POSICIÓN

Para obtener la posición actual del usuario se peude usar el método **getCurrrentPosition** que posee, como parámetro, una función callback que recibe un objeto de tipo **Position**. Mediante este objeto podemos conseguir las coordenadas GPS de la localización del usuario:

```
<body>
<main></main>
</body>
<script>
        var main=document.querySelector("main");
```

```
        navigator.geolocation.getCurrentPosition(pos=>{
                main.textContent=`latitud: ${pos.coords.latitude},` +
                        `Longitud: ${pos.coords.longitude}`;
        })
</script>
```

Este código permite acceder a la propiedad **coords** del objeto de posición y así conseguir las coordenadas de longitud (**longitude**) y latitud (**latitude**), las cuales se muestran en el cuerpo del documento.

Aun más interesante es utilizar esas coordenadas para pasarles (vía AJAX o con un enlace) a un servicio de Internet para, por ejemplo, que nos muestre un mapa centrado en esa posición. Es lo que hace el servicio **openstreetmap.org** que proporciona una API gratuita para poder utilizar sus mapas en nuestras páginas.

El objeto de posición proporciona estas propiedades:

PROPIEDAD	USO
speed	Velocidad (en metros por segundo) a la que se desplaza el dispositivo al que le estamos obteniendo la posición
altitude	Altura sobre el nivel del mar
accuracy	Precisión (en metros) de la longitud y la latitud
altitudeAccuracy	Precisión (en metros) de la altitud
heading	Ángulo en el que el dispositivo se está moviendo

Además, la fución **getCurrentPosition** acepta un segundo parámetro que es otra función callback que se ejecuta en caso de errores al obtener la posición del usuario. Esa función permite recoger el error.

5.2. OBTENER POSICIÓN CONTINUAMENTE

Hay otro método llamado **watchPosition** que funciona exactamente igual que **getCurrentPosition**. La diferencia es que watchPosition se invoca continuamente de forma automática para recalcular la posición, lo que le hace ideal como método de seguimiento de dispositivos móviles.

El método devuelve un identificador asociado al examinador de posición que invoca a este método continuamente. Si deseamos que se deje de invocar el método, entonces debemos usar el método **clearWatch**, el cual recibe el identificador obtenido. De esa forma se deja de examinar la posición del dispositivo.

```
<main></main>
</body>
<script>
        var main=document.querySelector("main");
        var cont=0;
        let id=navigator.geolocation.watchPosition(pos => {
                cont++;
```

```
                let nodo=document.createElement("p");
                nodo.textContent=
                        `Latitud: ${pos.coords.latitude},` +
                        `Longitud: ${pos.coords.longitude}`;
                main.appendChild(nodo);
                if (cont==3) navigator.geolocation.clearWatch(id);
        })
</script>
```

El código muestra la posición del dispositivo, pero veremos el texto 3 veces. **watchPosition** llama a la función callback continuamente para examinar la posición, pero hay un contador (cont) que cuando vale 3 hace que se invoque **clearWath** para dejar de examinar la posición.

6. API PARA MULTIMEDIA

El vídeo y audio son cada vez más un recurso más habitual en las aplicaciones web. En muchas ocasiones se utilizan vídeos procedentes de servicios de Internet como Youtube o vimeo que porporcionan un código especial para incrustar el vídeo en nuestras páginas de forma cómoda. El problema de esta forma de colocar vídeo es que dependemos de la API concreta del servicio y que, además, estamos sujetos a los intereses de estos servicios. Es decir, no podremos controlar la publicidad, los controles ni tampoco qué enlaces a otros vídeos se van a ofrecer a nuestros usuarios.

Hay otras ventajas que nos ofrece el uso de esos servicios, como el hecho de no ser nosotros los que gestionamos su almacenamiento, que tampoco nos preocuparemos por la velocidad a la que ofrecemos el vídeo (la delegamos a ese servicio) y, la más imporante, la visibilidad social que ofrecen.

Pero, cuando queremos mostrar un vídeo propio que queremos controlar al 100%, entocnes se utilizan las etiquetas video o audio para mostrarlo:

```
<video src=”video.mp4”>
        No se puede mostrar el vídeo en este navegador
</video>
```

Por defecto, no se ofrecen botones de control de vídeo. Si los deseáramos, disponemos de atributos:

```
<video src=”video.mp4” controls>
        No se puede mostrar el vídeo en este navegador
</video>
```

Aunque el código HTML es muy cómodo, cada navegador usa un formato de controles propio. En muchas ocasiones desearemos ser nosotros los que controlemos el vídeo.

Si queremos hacer un control más personal de los medios que incrustamos en las páginas, entocnes, podemos acudir a la API que nos proporciona JavaScript para este fin. Esa API se asocia a los elemnetos de tipo **video** o **audio** y desde JavaScript podremos acceder a los siguientes métodos y propiedades:

MÉTODO O PROPIEDAD	USO
play()	Reproduce el medio. Si estaba en pausa, se abandona la pausa

pause()	Pausa el medio
load()	Recarga el vídeo y coloca el cursor de posición de inicio
paused	Propiedad booleana de solo lectura (no se puede modificar) que indica si el vídeo está en pausa
loop	Valor booleano que, con valor true, indica que el vídeo se reproducirá en un bucle contínuo. Video.loop=true;
muted	Valor booleano que sirve para mostrar si el medio está silenciado o no. Permite cambiar el estado. Video.muted=true;
volume	Permite ver o modificar el valor del volumen del vídeo o audio al valor indicado. El valor es un número del 0 al 1: Video.volume=0.5;// Volumen a la mitad
controls	Booleano que permite indicar si deseamos mostrar o no los controles del navegador para pausar, retroceder, etc. Video.controls=false; // No deseamos controles
duration	Propiedad de solo lectura, en segundos, que devuelve el número de segundos que dura el vídeo.
currentTime	Valor decimal que muestra, en segundos, la posición actual en la que se reproduce el vídeo. Permite modificar dicha posición. Video.currentTime=30; //posiciona el vídeo a 30s
ended	Propiedad booleana de solo lectura, que permite saber si el vídeo ha terminado de reproducirse
playbackRate	Permite especificar o examinar la velocidad a la que se reproduce el vídeo o audio. Un valor de 1, es velocidad normal. Por debajo, velocidad lenta y por encima, velocidad rápida.

También hay eventos propios relacionados con la reproducción de medios que permiten realizar acciones avanzadas.

Así, este código nos permite mostrar por debajo del vídeo, el segundo en el que se está reproduciendo el vídeo, gracias a la captura del evento **timeupdate** que se lanza continuamente (cada vez que el vídeo cambia su cursor de posición temporal) cuando el vídeo se reproduce.

```
<body>
        <video src="POCOYO.mp4" controls>
```

```
        No se puede mostrar este vídeo en el navegador
        </video>
<main>Pruebas</main>
</body>
<script>
let video=document.querySelector("video");
let main=document.querySelector("main");
video.addEventListener("timeupdate",(ev)=>{
        main.textContent="Posición en segundos: "+ video.currentTime;
});
</script>
```

7. NOTIFICACIONES

(Nota: con archivo .htm las notificaciones puede que no se lancen. Usar .html)

7.1. FUNCIONAMIENTO DE LA API DE NOTIFICACIONES

En los últimos años ha crecido la necesidad de que las aplicaciones web se comuniquen con el usuario para informar de novedades, cambios o cualquier otra información. Esta necesidad procede, realmente, de lo que las aplicaciones móviles han utilizado desde hace muchos años: las notificaciones al usuario.

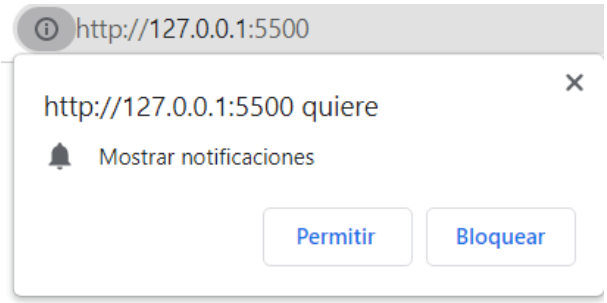
Se trata de mensajes que se envían al usuario y el sistema gestiona de manera especial. En el caso de las aplicaciones web suelen ser mensajes en una zona especial del navegador. Como ocurrirá con la API de Geolocalización, el usuario debe aceptar las notificaciones para que estas tengan lugar.

El objeto que controla las notificaciones se llama **Notification**. El permiso al usuario lo solicitaremos mediante un método estático **Notification.requestPermission**(). Este método proporciona una promesa que se resuelve positivamente cuando el usuario contesta a la petición. El objeto de respuesta guarda el texto "**granted**" si el usuario aceptó el mensaje de permiso o "**denied**" si denegó el permiso.

```
Notification.requestPermission()
```

```
.then(resp=>{
        if(resp=="granted")
                console.log(`Permiso concedido`);
        else
                console.log(`Permiso denegado`);
});
```

El código anterior muestra por consola si se concedió o no el permiso. Si recargamos la página, veremos que ya no nos pregunta más por las notificaciones. La mayoría de navegadores guardan nuestra respuesta durante toda la sesión. Además, suelen proporcionar (normalmente en la barra de direcciones) algún icono o propiedad que permita modificar de nuevo el estado de la notificación.

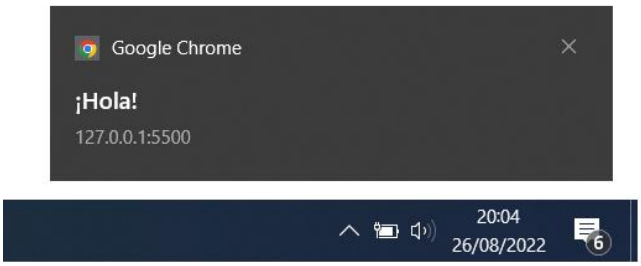


Generar una nueva notificación es crear un objeto de tipo **Notification**. En la construcción del objeto podemos indicar el texto que deseamos mostrar en la notificación.

Así pues, el código JavaScript que pida permiso de notificaciones y, en caso de que el usuario acceda, muestre un mensaje de saludo, sería el siguiente:

```
Notification.requestPermission()
.then(resp=>{
        if(resp=="granted")
        new Notification("¡Hola!");
});
```

La notificación, normalmente aparece en un área concreta del sistema. Por ejemplo, en los sistemas Windows suele ser el área inferior derecha donde está el área de notificaciones.



Para saber si disponemos de permisos de notificación antes de requerirlo al usuario, el objeto **Notification** dispone de una propiedad llamada **permission** la cual puede tomar los valores **denied** o **granted**. Pero si al usuario aún no se le ha pedido permiso, esta propiedad toma el valor **default**. De ese modo solo requeriremos el permiso si es necesario:

```
let permiso=Notification.permission;
if(permiso=="default"){
```

```
	Notification.requestPermission()
	.then(resp=>{
		if(resp=="granted")
		new Notification("¡Hola!"); //En firefox da error, revisar avisos
	});
}
```

El formato de las notificaciones puede ser más avanzado porque el constructor **Notification** admite 2 parámetros: el primero es el título y el segundo es un objeto que puede indicar 2 propiedades: **body** para el contenido de la notificación e **icon** para colocar un icono en la notificación:

```
new Notification("Bienvenida",{
	body:"Bienvenida/o a esta <strong>web</strong>",
	icon:"WebLogo.png"
})
```

Las notificaciones, normalmente, se cierran automáticamente por parte del sistema. Pero el método **close**(), nos permite cerrarlas desde el propio código:

```
let permiso=Notification.permission;
if(permiso=="default"){
	Notification.requestPermission()
	.then(resp=>{
		if(resp=="granted")
			new Notification("¡Hola!");
		setTimeout(()=>n.close(),2000);
	});
}
```

El código anterior cierra la notificación a los 2 segundos.

Las notificaciones, como cualquier elemento del DOM, disponen de eventos. Por ejemplo, podemos hacer que, al hacer clic sobre la notificación, abramos una URL concreta:

```
function mostrarNotificacion(){
	let n=new Notification("Información",{
	  body: "¡Puedes ira a mi página si haces clic"
	});
	n.addEventListener("click",(ev)=>{
	  window.open("https://www.beniarjo.es");
	});
}

let permiso=Notification.permission;
if(permiso=="default"){
	Notification.requestPermission()
	.then(resp=>{
		if(resp=="granted")
			mostrarNotificacion();
	})
}
```

```
else if(permiso=="granted"){
        mostrarNotificacion();
}
```

7.2. CONCLUSIONES SOBRE LAS NOTIFICACIONES

La API de notificaciones es bastante reciente y, además, la configuración del usuario en el sistema operativo permite, fácilmente, su deshabilitación. Es un recurso muy interesante para informaciones anexas a la principal que muestra una aplicación web para avisos que queremos que se vean fuera del navegador.

Un ejemplo de aviso podría es el que proponemos en la práctica, un formulario de suscripción a un servicio en el que hemos detectado que el usuario no ha enviado. Aunque la página web esté en segundo plano, porque el usuario se ha distraído con otro software de su sistema, podemos enviarle una notificación por ejemplo a los 15 segundos, indicando que aún no ha enviado los datos.

En todo caso, no hay que abusar de este recurso. El abuso de las notificaciones genera un rechazo por parte de los usuarios.

Nota: En Chrome revisar el estado de las notificaciones en: