Jorge García Flores

Desarrollo Web en Entorno Cliente

Tema 16

API WEB

API Web

Con HTML5, JS pone a disposición del programador un gran número de APIs que permiten realizar muchas tareas de forma sencilla.

Estas APIs son conocidas, en su conjunto, como API Web o API nativa de JS, y no es necesario importar o cargar librerías para usarlas.

Para obtener una lista de todas las API y más información de su uso se recomienda consultar la siguiente documentación:

https://developer.mozilla.org/es/docs/Web/API

A continuación, se hace un repaso de las APIs más útiles que se pueden usar al desarrollar una aplicación o sitio Web.

Drag and Drop Web

La API *Drag and Drop* (arrastrar y soltar) permite a los usuarios hacer clic y mantener presionado el botón del ratón sobre un elemento arrastrándolo de una ubicación a otra.

En la mayoría de los navegadores, el texto seleccionado, las imágenes y los enlaces se pueden arrastrar de forma predeterminada.

Esta API hace uso de la interfaz *`DragEvent`* que hereda propiedades de las interfaces *`MouseEvent`* y *`Event`*:

Para hacer que un elemento se pueda arrastrar se le añade el atributo *`draggable=true`*. Se puede habilitar la acción de arrastrar en casi cualquier cosa: imágenes, enlaces, archivos,…

```
<div class="container">
   <div draggable="true" class="box">A</div>
</div>
```

Drag and Drop Web

La API *Drag and Drop* (arrastrar y soltar) dispone de varios eventos, cuando se arrastra un elemento se activan estos eventos en la siguiente secuencia:

`e.target` representa el elemento que se está arrastrando en los siguientes eventos.

- *dragstart*. Cuando se mantiene presionado el botón del mouse y se comienza a mover elelemento se activa el evento *dragstart*. El cursor cambia a un símbolo de no soltar (un círculo con una línea que lo atraviesa) para indicar que no puede soltar el elemento sobre sí mismo.

- *drag*. Después de dispararse el evento *dragstart*, se activa el evento *drag* cada vez que se arrastra el elemento.

- *dragend*. Este evento se activa cuando deja de arrastrarse el elemento.

Drag and Drop Web

Cuando se arrastra un elemento sobre un elemento válido se activan estos eventos en la siguiente secuencia:

e.target representa el elemento destino, el elemento válido que se encuentra bajo el elemento que se está arrastrando.

- *dragenter*. Este evento se activa tan pronto como un elemento se arrastra sobre un elemento destino de la colocación.

- *dragover*. Se activa cada vez que se arrastra el elemento sobre el área del elemento de destino de la colocación. Mientras el elemento esté sobre el elemento de destino continuará activándose el evento.

- *dragleave o drop*. Cuando se arrastra el elemento fuera del área del elemento de destino se deja de activar el evento *dragover* y se activa *dragleave*. Cuando se arrastra y se suelta el elemento en el área del elemento de destino se deja de activar el evento *dragover* y se activa el evento *drop*.

Drag and Drop Web

Para transferir datos en una acción de arrastrar y soltar se utiliza el objeto *dataTransfer*.

El objeto *dataTransfer* es una propiedad del evento y permite transferir datos desde el elemento arrastrado al elemento destino.

El objeto *dataTransfer* tiene los métodos: *clearData()*, *setData()* y *getData()*.

El método *clearData()* elimina la data desde el elemento arrastrado, si no hay datos no hace nada. Solo se puede utilizar en el evento *dragstart*.

```
clearData();
clearData(format);
```

Recibe como argumento una cadena que especifica el tipo de datos que se quiere eliminar. Si este parámetro es una cadena vacía o no se proporciona, se eliminan los datos de todos los tipos. Este método no elimina archivos, por lo que es posible que todavía haya datos de tipo "*Files*" en la lista de objetos *DataTransfer.types* si se están arrastrando ficheros del escritorio al navegador.

Drag and Drop Web

El método *setData()* permite especificar los datos que se desean transferir y en que formato:

```
dataTransfer.setData(format, data)
```

El formato normalmente será *text/plain* o *text/uri-list*, y los datos se envían en forma de cadena, por lo que puede personalizarse, serializando un objeto literal en formato JSON.

El método *getData()* permite recuperar los datos transferidos en el elemento destino, para eso hay que especificar el mismo formato en el que se han enviado:

```
dataTransfer.getData(format)
```

Drag and Drop Web

En el primer ejemplo, se envía el *id* del elemento arrastrado en el *dataTransfer*, lo que permitirá al elemento destino acceder a dicho elemento.

```
element.addEventListener("dragstart",(e) => {
    e.dataTransfer.clearData();
    e.dataTransfer.setData("text/plain", e.target.id);
});
```

En el ejemplo siguiente se construye un objeto literal para enviar un dato personalizado en el *dataTransfer*.

```
element.addEventListener("dragstart",(e) => {
    const foo={
        nombre: "Jorge",
        apellidos: "Garcia"
    };
    e.dataTransfer.clearData();
    const fooJSON = JSON.stringify(foo);
    e.dataTransfer.setData("foo", fooJSON);
});
```

Drag and Drop Web

La data del objeto *dataTransfer* no es accesible desde los eventos *dragenter*, *dragover y dragleave*, por lo que si se consulta desde estos eventos se obtendrán datos vacíos.

La data del objeto *dataTransfer* se genera en el evento *dragstart* cuando se hace clic y se arrastra el elemento y se consulta cuando se dispara el evento *drop*, al soltar el elemento arrastrado sobre el elemento destino.

No obstante, algunos navegadores, como, por ejemplo, Firefox, puede dar acceso a la data del objeto *dataTransfer* fuera del evento *drop*, pero no es el comportamiento estándar, por lo que se desaconseja aprovechar esta característica para evitar que la aplicación funcione mal dependiendo del navegador.

Drag and Drop Web

Ejemplos:

```
element.addEventListener("drop",(e) => {
   const idElement =e.dataTransfer.setData("text/plain");
});
```

```
element.addEventListener("drop",(e) => {
   const data = dataTransfer.getData("foo");
   const foo = JSON.parse(data);
});
```

Drag and Drop Web

El método *setDragImage(imgElement, x, y)* permite usar una imagen distinta a la imagen que se genera automáticamente cuando se arrastra un elemento. x e y representan un desplazamiento de la imagen en dichas coordenadas con respecto al cursor del ratón, siendo el origen *(0,0)* la esquina superior izquierda de la imagen.

En el siguiente ejemplo se usará la imagen con el id *circle* cuando se haga clic y se arrastre el elemento *element*, la imagen aparecerá 20 unidades desplazada hacia arriba y hacia la izquierda del cursor del ratón:

```
element.addEventListener("dragstart",(e) => {
    const img = document. querySelector("#circle");
    e.dataTransfer.setDragImage(img, 20, 20);
}
```

Drag and Drop Web

El objeto *dataTransfer* también dispone de una serie de propiedades que proporcionan información visual al usuario durante el proceso de arrastre y, sobre todo, controlan cómo responde cada elemento al ser arrastrado y soltado sobre otro elemento.

La propiedad *effectAllowed* especifica el efecto permitido y solo puede establecerse durante el evento *dragenter* a uno de los siguientes valores: *none*, *copy*, *copyLink*, *copyMove*, *link*, *linkMove*, *move*, *all* y *uninitialized*.

- La operación *de copia* se utiliza para indicar que los datos que se arrastran se copiarán desde su ubicación actual a la ubicación de colocación.
- La operación *de movimiento* se utiliza para indicar que los datos que se arrastran se moverán.
- La operación *de enlace* se utiliza para indicar que se creará algún tipo de relación o conexión entre las ubicaciones de origen y de colocación.

Drag and Drop Web

effectAllowed puede tener uno de los siguientes valores:

- *none* el elemento no se puede dejar caer.
- *copy* se puede hacer una copia del elemento en la zona destino.
- *copyLink* se permite una operación de copia o enlace.
- *copyMove* se permite una operación de copiar o mover.
- *link* se puede establecer un vínculo con la fuente en la nueva ubicación.
- *linkMove* se permite una operación de enlace o movimiento.
- *move* un artículo puede trasladarse a una nueva ubicación.
- *all* todas las operaciones están permitidas.
- *uninitialized* el valor por defecto cuando no se establece, equivale a *all* .

Drag and Drop Web

La propiedad *dropEffect* se establece sobre el elemento de desino durante los eventos *dragenter* y *dragover*, especifica la operación que recibe el elemento de destino. Puede tener los siguientes valores: *none*, *copy*, *move* y *link*.

Por ejemplo, un elemento arrastrado con un valor de *copy* en *effectAllowed* no puede soltarse o disparar el evento *drop* de un elemento de destino con un valor de *move* en su propiedad *dropEffect*.

Además, afecta al tipo de cursor que se muestra mientras se arrastra. Por ejemplo, cuando un elemento se arrastra sobre otro elemento de destino el cursor del navegador puede indicar qué tipo de operación se producirá.

Drag and Drop Web

En el siguiente ejemplo el elemento *draggable* puede ser movido al elemento *dropzone* porque su *effectAllowed* es *copyMove* y porque el elemento *dropzone* tiene *dropEffect* es *move*:

```
draggable.addEventListener("dragstart", (event)=> {
  event.dataTransfer.setData('text/plain', event.target.id);
  event.currentTarget.style.backgroundColor = 'yellow';
  event.dataTransfer.effectAllowed = copyMove;
});

dropzone.addEventListener("dragenter", (event)=> {
  event.dataTransfer.dropEffect = move;
  event.preventDefault();
});

dropzone.addEventListener("drop", (event)=> {
  const id = event.dataTransfer.getData('text');
  const draggableElement = document.getElementById(id);
   event.currentTarget.appendChild(draggableElement);
});
```

Drag and Drop Web

La propiedad de solo lectura *items* es una lista de los elementos de datos en una operación de arrastre. La lista incluye un elemento por cada elemento de la operación y, si la operación no tenía elementos, la lista está vacía.

La propiedad *types* de solo lectura devuelve los tipos disponibles que existen en la propiedad *items*.

La propiedad *files* representa una lista de archivos que están siendo arrastrados. Si la operación no incluye archivos, la lista está vacía. Esta función se utiliza para arrastrar archivos desde el escritorio de un usuario al navegador. En el siguiente ejemplo mostrará en la consola del navegador los nombres de los ficheros junto con su tipo y tamaño cuando se dejen caer sobre el elemento *element*:

```
element.addEventListener("drop",(e) => {
   files = event.dataTransfer?.files ?? event.target.files;
   let len = files.length;
   for (let i=0; i<len; i++)
     console.log(`${files[i].name} (${files[i].type}, ${files[i].size} bytes)`;}
```

intersection observer

La API *Intersection Observer* permite "*observar*" de forma asíncrona cuando un elemento, como el *viewport,* se cruza con otro, interseca, un elemento observado y un elemento superior, o el propio *viewport*.

Un *viewport* representa la región poligonal, normalmente rectangular, que está siendo visualizada en ese instante. En los navegadores web, se refiere a la parte del documento que se está viendo, la que está visible actualmente en la ventana o la pantalla, si el documento está siendo visto en modo pantalla completa. El contenido fuera del *viewport* no es visible en la pantalla hasta que se llega a él, normalmente haciendo *scroll*.

Mediante esta API se pueden cargar imágenes de forma *lazy* (perezosa), implementar el *scroll infinito* o mostrar anuncios al llegar a determinadas partes de la web.

Lazy load (carga perezosa) mejora el rendimiento de los sitios al evitar cargar imágenes que no estarán visibles inicialmente. Aunque, con el nuevo estándar HTML5 se puede hacer *lazy load* nativo de las imágenes e `iframes` gracias al atributo `Loading` de HTML, sin necesidad de utilizar JS.

intersection observer

Intersection Observer permite configurar una función *callback* que es llamada cuando se produce intersección entre un elemento observado y un elemento superior, conocido con el nombre de `root`.

Para utilizar esta API hay que crear un objeto `IntersectionObserver` a través de su constructor.

La sintaxis es la siguiente:

```
const observer=new IntersectionObserver(callback, options);
```

El constructor recibe dos argumentos:

- ***callback*** que indica la función que será ejecutada cuando el umbral que se especifique en el parámetro `options` sea traspasado.

- ***options*** un parámetro opcional que representa un objeto que especifica la configuración del `observer`.

intersection observer

El argumento opcional *`options`* permite, a través de las propiedades de un objeto literal, especificar las siguientes características:

`root` especifica el elemento que deberá emplear el *observer* para comprobar la intersección. Por defecto es el *viewport* del navegador.

`rootMargin` especifica el margen alrededor del elemento *`root`* de modo que se pueda alargar o estrechar los lados del elemento *`root`* a la hora de realizar la comprobación. Su valor se establece como si fuera una propiedad CSS, por ejemplo: *`2px 3px 1px 6px (top, right, bottom, left)`*.

`threshold` especifica en qué porcentaje de la visibilidad del elemento observado se debe ejecutar la *callback*. Por ejemplo, si se quiere detectar cuando se ha traspasado el 50% se especifica un *`threshold`* con valor *`0.5`*, en cambio, un valor de *`1.0`* indica que el umbral no se considera traspasado hasta que todos los píxeles del elemento sean visibles. Se puede proporcionar mediante un número o un array de números, en el caso del array permite llamar al *callback* cada vez que se alcance un porcentaje de visibilidad, por ejemplo, si se quiere llamar cada 25% sería *`[0, 0.25, 0.5, 0.75, 1]`*.

intersection observer

Ejemplo:

```
const options = {
  root: document.querySelector('#main-container'),
  rootMargin: '10px 0px',
  threshold: 1.0
};

const observer=new IntersectionObserver(callback, options);
```

IntersectionObserver dispone de varios métodos:

- ***disconnect()*** hace que deje de observarse un elemento que se le pasa como argumento.
- ***takeRecords()*** devuelve un array de objetos *IntersectionObserverEntry* con los elementos a observar por ese *observer*.

observe

El método *observe* añade un elemento al conjunto de elementos que supervisa el *IntersectionObserver*.

Si se necesita observar varios elementos, se recomienda observarlos usando una única instancia *IntersectionObserver* llamando varias veces al método *observe()*.

A todos los elementos observados mediante un ***observer*** se les aplicará los parámetros de configuración definidos, o sus valores por defecto, en caso de que no se hayan especificado.

La sintaxis es:

```
observer.observe(targetElement);
```

targetElement es el nuevo elemento a observar. Su valor se selecciona mediante la API del DOM, y debe ser un descendiente del elemento raíz o estar contenido dentro del documento actual, si la raíz es el *viewport* del documento.

observe

Cuando un elemento a observar cruza uno de los umbrales de visibilidad del observador se invoca la función *callback* que se especificó en el constructor cuando se creó el `observer`.

A esta función se le pasa una matriz de objetos de tipo `IntersectionObserverEntry`, que representa cada uno los elementos a observar.

Los objetos `IntersectionObserverEntry` disponen, entre otras, de las siguientes propiedades:

`isIntersecting` un valor booleano que contiene `true` si el elemento interseca con el elemento raíz del observador, `false` en caso contrario.

`target` contiene una referencia al elemento a observar.

Se recomienda visitar la siguiente página para obtener más información sobre el objeto `IntersectionObserverEntry`:

https://developer.mozilla.org/en-US/docs/Web/API/IntersectionObserverEntry

unobserve

El método *unobserve* elimina un elemento del conjunto de elementos que supervisa el *IntersectionObserver*.

La sintaxis es:

```
observer.unobserve(targetElement);
```

targetElement es el elemento a dejar de observar.

Ejemplo:

```
const observer = new IntersectionObserver(entries=>{
  if(entries[0].isIntersecting){
    console.log('Ya esta visible en... ', entries[0].target);
    if (entries[0].isIntersecting){
      observer.unobserve(document.querySelector(`#${entries[0].target.id}`));
      mostrarVideo(`#${entries[0].target.id}`;}}});
```

navigator

La interfaz *`Navigator`* representa el estado y la identidad del navegador, también llamado más formalmente como *user agent*.

Proporciona mucha información relativa al usuario, como el navegador que se está usando, el lenguaje que está configurado,…

Además, da acceso a mucha funcionalidad, como, por ejemplo, saber si se accedió a la página o fue refrescada, la geoposición, el estado de la conexión…

Se recomienda visitar la siguiente página para obtener más información sobre el objeto *`Navigator`*:

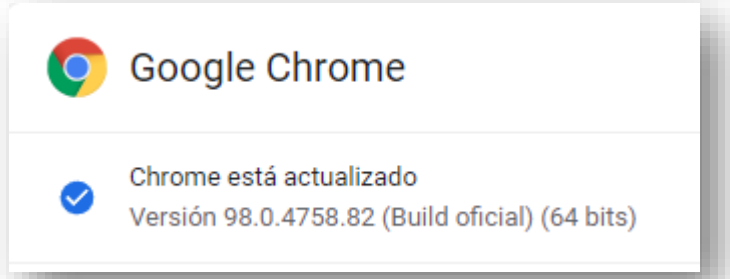
https://developer.mozilla.org/es/docs/Web/API/Navigator

A continuación, se muestran algunas de las muchas propiedades y métodos que proporciona.

navigator

userAgent es una propiedad de solo lectura que devuelve, en formato string, la cadena el agente de usuario del navegador, a partir de ella se puede obtener el navegador y su versión.

```
navigator.userAgent
'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrom
e/98.0.4758.82 Safari/537.36'
```



Language devuelve una cadena que indica el idioma preferido del usuario, generalmente el idioma de la interfaz de usuario del navegador. Un valor de *null* significa que no se ha podido acceder al idioma o es desconocido.

```
navigator.language
'es-ES'
```

navigator

online es una propiedad de solo lectura que devuelve un booleano indicando si hay conexión o no a Internet.

La propiedad se actualiza cada vez que cambia la capacidad del navegador para conectarse a la red.

La actualización ocurre cuando el usuario hace clic en enlaces o cuando un script solicita una página remota. Por ejemplo, la propiedad se volverá *false* cuando el usuario haga clic en un enlace poco después de perder la conexión a Internet.

Se puede utilizar los eventos *offline* y *online* sobre el objeto *window* para actuar sobre esta situación.

Ejemplo:

```
window.addEventListener('offline',(e)=>{console.log('No hay Internet')});
window.addEventListener('online',(e)=>{console.log('Ya hay Internet')});
```

fetch

Hasta el momento, para hacer peticiones HTTP se dispone del objeto *`XMLHttpRequest`* y del método Ajax de la librería jQuery, este último era el preferido debido a que simplificaba mucho el proceso. Recientemente, se ha incorporado una nueva API llamada *Fetch*.

Fetch permite, con código más sencillo, realizar peticiones HTTP asíncronas utilizando promesas.

Sin embargo, la especificación *Fetch* presenta dos diferencias con respecto a Ajax de la librería jQuery:

- La promesa devuelta por *Fetch* no será rechazada con estados de error HTTP, incluso si la respuesta es un error *404* o *500*; en estos casos se resuelve normalmente, pero con un estado *ok* configurado a *`false`*, y la promesa solo se rechaza con un fallo de la red o algo similar que impide completar la solicitud.

- Por defecto, *Fetch* no envía ni recibe cookies del servidor, realizando peticiones no autenticadas si el sitio permite mantener una sesión de usuario, pero, si se requiere, se pueden mandar cookies y credenciales configurándolo a través de sus opciones.

fetch

La sintaxis más sencilla para realizar una petición HTTP asíncrona es llamar al método *fetch* y pasarle por parámetro la petición URL:

```
peticion=fetch(url)

peticion
.then(function(response) { ... })
.catch(function(error) { ... })
```

Cuando se resuelve la petición, la promesa será resuelta y se devolverá un objeto *Response*, la función del *then* servirá para procesarla.

Si surge algún error, un fallo de la red o algo similar que impide completar la solicitud, se puede procear mediante el bloque del *catch*.

fetch

Como el método *fetch* es estático suele escribirse de una forma más simple:

```
fetch(url)
.then(function(response) { ... })
.catch(function(error) { ... })
```

Además, a *then* se le pasa una función *callback*, donde su parámetro *response* es el objeto de respuesta de la petición que se ha realizado, y que procesa la respuesta a la petición.

Cuando se necesita configurar opciones adicionales a la petición HTTP se utiliza el parámetro opcional que le sigue a la url, este parámetro es un objeto literal que contine la configuración deseada:

```
fetch(url, opciones)
.then(function(response) { ... })
.catch(function(error) { ... })
```

fetch

En el parámetro opciones, que recibe el método *fetch* para realizar una petición HTTP, se corresponde con un objeto en el que se puede definir:

Propiedad	***Descripción***
method	Método HTTP que se emplea para realizar la petición. Por defecto, *GET*. Otras opciones disponibles: *HEAD*, *POST*, *PUT*, *DELETE*...
body	Cuerpo de la petición, puede ser de varios tipos: *String*, *FormData*, *Blob*...
headers	Permite especificar las cabeceras HTTP mediante un objeto literal o un objeto *Headers*.
credentials	Modo de credenciales. Por defecto, *omit*. Otras opciones: *same-origin* e *include*.

credentials permite modificar el modo en el que se realiza la petición. El valor *omit* hace que no se incluyan credenciales en la misma, pero es posible indicar: *same-origin*, que incluye las credenciales si se está en el mismo dominio, o *include* que incluye las credenciales incluso en peticiones a otros dominios.

En el caso de realizar peticiones a dominios diferentes, si el servidor no está configurado, se obtendrá un problema de CORS (*Cross-Origin Resource Sharing*).

fetch

Cross Origin (origen cruzado) es la palabra que se utiliza para denominar el tipo de peticiones que se realizan a un dominio diferente del dominio de origen desde donde se realiza la petición. De esta forma, por una parte, hay peticiones de origen cruzado (*cross-origin*) y peticiones al mismo origen (*same-origin*).

CORS (*Cross-Origin Resource Sharing*) es un mecanismo o política de seguridad que permite controlar las peticiones HTTP asíncronas que se pueden realizar desde un navegador a un servidor con un dominio diferente de la página cargada originalmente. Este tipo de peticiones se llaman peticiones de origen cruzado (*cross-origin*).

Ejemplo de configuración para realizar una petición POST:

```
const options = {
  method: "POST",
  headers: {"Content-Type": "application/json"},
  body: JSON.stringify(jsonData)
};
fetch(url, options);
```

fetch

Ejemplo:

```
const apiKey="245af25cc4f2b1ded6d80044e37c5596";
const cityName="Salamanca,ES";
const url=`https://api.openweathermap.org/data/2.5/weather?q=${cityName}&appid=${apiKey}`;
fetch(url)
.then((resp) => resp.json())
.then(console.log);
```

Puede observarse cómo hay dos métodos *then()* en el código, esto se debe a que lo que devuelve *fetch* es una promesa, que se atrapa con el primer *then*, pero también lo que devuelve *resp.json()* es otra promesa, que es atrapada con el segundo *then* (a través de *method chaining)*.

El método *json()* es un método del objeto *Response* que devuelve una promesa, la promesa se resuelve al analizar el cuerpo de la respuesta como un JSON, el resultado ya no es un JSON sino el resultado de tomar el JSON como entrada y analizarlo para producir un objeto literal de JS.

response

Response es una interfaz de la API *Fetch* que representa la respuesta a una petición. Se puede crear un nuevo Response mediante el constructor *`Response()`* .

Pero se suele utilizar más frecuentemente como el objeto devuelto de una operación de la API, por ejemplo, mediante una petición realizada a través del método *`fetch()`*.

A través de las propiedades del objeto *`Response`* se tiene acceso a mucha información de la petición:

`status` es una propiedad de solo lectura que contiene el código de estado de la respuesta, como, por ejemplo, 200.

`statusText` es una propiedad de solo lectura que contiene el mensaje de estado correspondiente al código de estado de la propiedad *`status`*. Por ejemplo, esto sería *`OK`* para un código de estado *`200`*, *`Continue`* para *`100`*, *`Not Found`* para *`404`*…

response

body es una propiedad de solo lectura que contiene el contenido del cuerpo de la respuesta.

bodyUsed es una propiedad de solo lectura que contiene un valor booleano indicando si ya se ha accedido al cuerpo de la respuesta.

headers es una propiedad de solo lectura que contiene un objeto *Headers* asociado a la respuesta.

ok es una propiedad de solo lectura que contiene un valor booleano que indica si la respuesta tuvo éxito, un estado entre 200 y 299, o no.

redirected es una propiedad de solo lectura que indica si la respuesta es o no el resultado de una redirección, es decir, si su lista de URL tiene más de una entrada.

type es una propiedad de solo lectura que contiene el tipo de respuesta, como, por ejemplo, *basic*, *cors*,…

response

url es una propiedad de solo lectura que contiene la URL de la respuesta.

trailers es la resolución a promesa asociada al uso de cabecera particular.

El objeto *Response* también dispone de métodos que permiten, entre otras cosas, acceder al texto de la respuesta:

text() devuelve una promesa que se resuelve al formatear el cuerpo de la respuesta en texto.

json() devuelve una promesa que se resuelve con el resultado al analizar el texto del cuerpo de la respuesta como si fuera un JSON, devuelve un objeto literal de JS.

blob() devuelve una promesa que se resuelve al representar en formato *Blob* el cuerpo de la respuesta.

response

formData() devuelve una promesa que se resuelve al representar como un *FormData* el cuerpo de la respuesta.

error() devuelve un nuevo objeto *Response* asociado a un error de red.

clone() devuelve un clon del objeto *Response* recibido.

arrayBuffer() devuelve una promesa que se resuelve al representar en formato *ArrayBuffer* (buffer binario puro) el cuerpo de la respuesta.

redirect(url, code) redirige a una URL diferente, opcionalmente con un *code* de error.

response

Ejemplo:

```
const imagen=document.querySelector('.logo');
fetch('logo.jpg')
.then(response=>response.blob())
.then(blob=>{
  const objectURL=URL.createObjectURL(blob);
  imagen.src=objectURL;
});
```

headers

Headers es un objeto útil para trabajar con cabeceras. Se crea un objeto de este tipo a través del constructor *Headers()*:

```
const headers = new Headers();
headers.set("Content-Type", "application/json");
```

Dispone de los siguientes métodos útiles:

Método	***Descripción***
has(name)	Comprueba si la cabecera *name* está definida.
get(name)	Obtiene el valor de la cabecera *name*.
set(name, value)	Establece o modifica el valor *value* a la cabecera name.
append(name, value)	Añade un nuevo valor *value* a la cabecera *name*.
delete(name)	Elimina la cabecera *name*.

Además, como muchos otros objetos iterables, se puede utilizar los métodos *entries()*, *keys()* y *values()* para recorrerlo.

notification

La API Web de JS permite enviar notificaciones a los usuarios. Estás notificaciones solo están disponibles a través de HTTPS y son una forma de advertir al usuario de nueva información de interés de un sitio web de manera asincrónica.

Para poder enviar notificaciones previamente los usuarios deben de haberlas aceptado.

Para crear una nueva notificación se usa la interfaz *`Notification`* a través de su constructor.

La sintaxis es la siguiente:

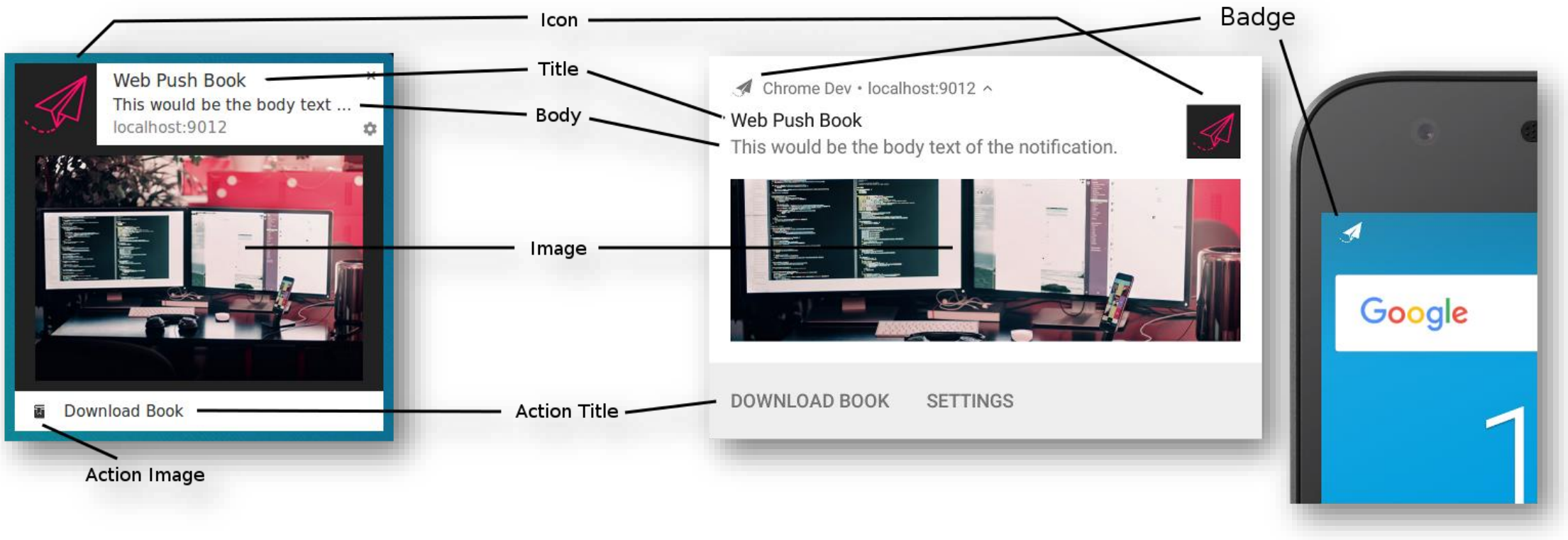
```
const notificacion = new Notification(título, opciones);
```

`título` define un título para la notificación, que será mostrado en la parte superior de la ventana de la notificación.

`opciones` es un argumento opcional y se corresponde con un objeto con la configuración que se quiere aplicar a la notificación.

notification

A continuación, se muestran los distintos elementos de presentes en una notificación que se pueden configurar mediante el argumento *opciones*.



notification

Las posibles opciones que se pueden aplicar a través del objeto que define la configuración son:

dir indica la dirección en la que se mostrará el texto de la notificación. Por defecto, es *auto*, lo que se ajustará al comportamiento definido en el navegador, otros valores son *rtl* y *ltr* que se corresponden con de derecha a izquierda o de izquierda a derecha respectivamente.

lang The El lenguaje usado para la notificación, de acuerdo a la RFC 5646 (consultar https://www.sitepoint.com/iso-2-letter-language-codes/).

badge una cadena de texto que indica la URL de la imagen usada para representar la notificación cuando no hay espacio suficiente para mostrarla en la ventana que contiene a la misma.

body una cadena representando el texto de la notificación, que será mostrado debajo del título.

tag una cadena que representa una etiqueta de identificación para la notificación.

notification

icon una cadena de texto que indica la URL de un icono que será mostrado en la en la notificación.

image: una cadena de texto que indica la URL de una imagen que será mostrado en la en la notificación.

data: datos, de cualquier tipo, que se quieren asociar a la notificación.

vibrate el tipo de vibración que si se quiere emitir mediante el hardware del dispositivo (móvil) con la notificación. Consultar la siguiente página para más información:

https://developer.mozilla.org/en-US/docs/Web/API/Vibration_API#vibration_patterns

renotify un valor booleano que indica si una nueva notificación reemplaza a otra más antigua. Por defecto es false, que indica que el usuario no quiere reemplazar las notificaciones antiguas con las nuevas.

notification

requireInteraction indica si la notificación permanece activa, valor *true*, hasta que el usuario la descarta o hace clic sobre ella, o si se cierra de manera automática, valor *false* que se corresponde con el valor por defecto.

actions es un array de acciones para mostrar en la notificación. Cada elemento del array es un objeto con lo siguiente:

- ***action***: una cadena que identifica la acción de usuario, mediante esta identificación se puede distinguir que acción fue activada en un evento.
- ***title***: una cadena que contiene el texto de la acción para mostrar al usuario.
- ***icon***: una cadena de texto que indica la URL de un icono que se quiere mostrar con la acción.
- ***silent*** un valor booleano que indica si la notificación es silenciosa, sin sonidos o vibraciones, a pesar de la configuración del dispositivo. Por defecto es false, que significa que no será silenciada.

notification

Para hacer un uso real de las notificaciones se necesita de funcionalidad programada en el lado del servidor, ya que desde el mismo se enviarán de manera periódica notificaciones a todos los usuarios suscritos, a través de lo que se conoce como *notificaciones push*.

La ventaja de las notificaciones es que no es necesario visitar la página para recibir notificaciones desde el servidor, esto se consigue mediante un *service worker.*

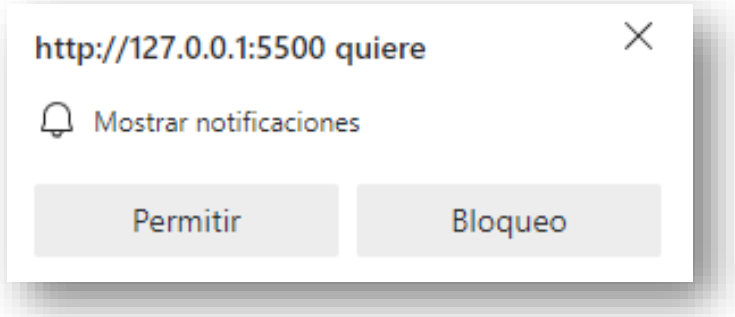
Para hacer uso, por ejemplo, de las acciones de una notificación, que enriquecen visual y funcionalmente las mismas, es necesario que la notificación se lance mediante un *service worker* (que se presentará a continuación).

Antes de poder enviar notificaciones al usuario este debe de haber aceptado recibirlas:

```
Uncaught TypeError: Failed to construct
'Notification': Actions are only supported
for persistent notifications shown using
ServiceWorkerRegistration.showNotification().
```

notification

Cuando se visita un sitio web que envía notificaciones se mostrará una ventana como la siguiente en la se solicita al usuario permiso para poder enviarle notificaciones:



Si el usuario lo permite se guardará en la configuración del navegador la posibilidad de recibir notificaciones. La decisión del usuario no se puede alterar desde código JS, solo desde la configuración del navegador. Pero sí se puede comprobar si el usuario acepta o no notificaciones mediante la propiedad estática, de solo lectura, *`Notification.permission`*.

notification

Los tres valores posibles de la propiedad *Notification.permission* son:

denied el usuario rechaza que las notificaciones sean mostradas.

granted el usuario acepta que las notificaciones sean mostradas.

default la elección del usuario es desconocida, por lo tanto, el navegador actuará como si el valor fuese *denied*.

Para solicitar permiso al usuario, con el fin de enviarle notificaciones, se utiliza el método estático *Notification.requestPermission()*que devuelve una promesa.

Ejemplo:

```
if(Notification.permission != 'granted') {
  Notification
   .requestPermission()
   .then(resultado => {console.log('El resultado es ', resultado)});}
```

notification

Las propiedades que posee un objeto de *`Notification`* son de solo lectura y permiten consultar los parámetros de la configuración de la notificación, por lo que se corresponden con las diferentes opciones de configuración de una notificación que se acaban de presentar. Por citar algunas:

badge que contiene la URL de la imagen usada para representar la notificación cuando no hay espacio suficiente para mostrarla.

`title` que se corresponde con el título de la notificación como está especificado en el parámetro opciones del constructor.

body el texto que aparecerá en el cuerpo de la notificación como está especificado en el parámetro opciones del constructor.

Se recomienda visitar la siguiente página para conocer el resto de las propiedades disponibles:

https://developer.mozilla.org/es/docs/Web/API/notification

notification

Además, se pueden tratar los siguientes eventos sobre la notificación:

- ***show*** es disparado cuando la notificación se muestra.
- ***click*** es disparado cuando se hace clic con el ratón sobre la notificación.
- ***close*** es disparado cuando la notificación se descarta o es cerrada con el método *close()*.
- ***error*** es disparada cuando ocurre un error, lo cual impide que la notificación sea mostrada.

Para manejar estos eventos se recomienda usar *addEventListener()*, aunque pueden utilizarse otros métodos como *onclick()* y *onerror()*, de ninguna manera se recomienda *onshow()* y *onclose()* ya que están obsoletos y podrían no funcionar en todos los navegadores.

Por último, se dispone del método estático *close()* que cierra una notificación y puede ser útil antes de mostrar una nueva notificación posterior.

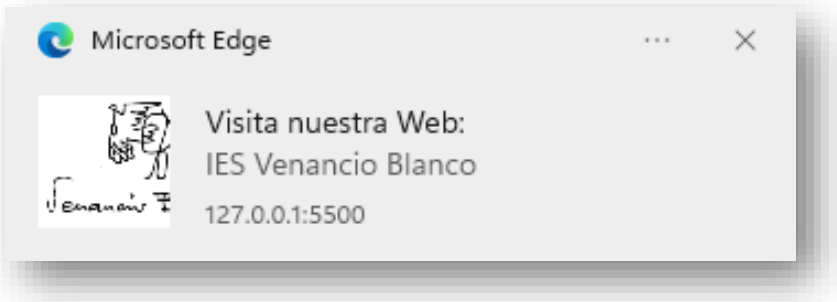
notification

El siguiente ejemplo muestra una notificación, y al hacer clic cobre la misma abrirá una ventana en el navegador a la URL *https://www.iesvenancioblanco.es/*:

```
const title='Visita nuestra Web:';
const options= {
   icon: 'img/icono.png',
   body: 'IES Venancio Blanco',
};

const notificacion=new Notification(title, options);

notificacion.addEventListener("click",()=> {
    window.open("https://www.iesvenancioblanco.es/");
});
```



service worker

Un *service worker* es un conjunto de instrucciones que se ejecutan en el navegador en segundo plano, separado de una página web, abriéndoles la puerta a funciones que no necesitan una página web ni la interacción de usuario.

Realmente es un fichero JS que continúa ejecutándose, aunque el sitio web esté cerrado.

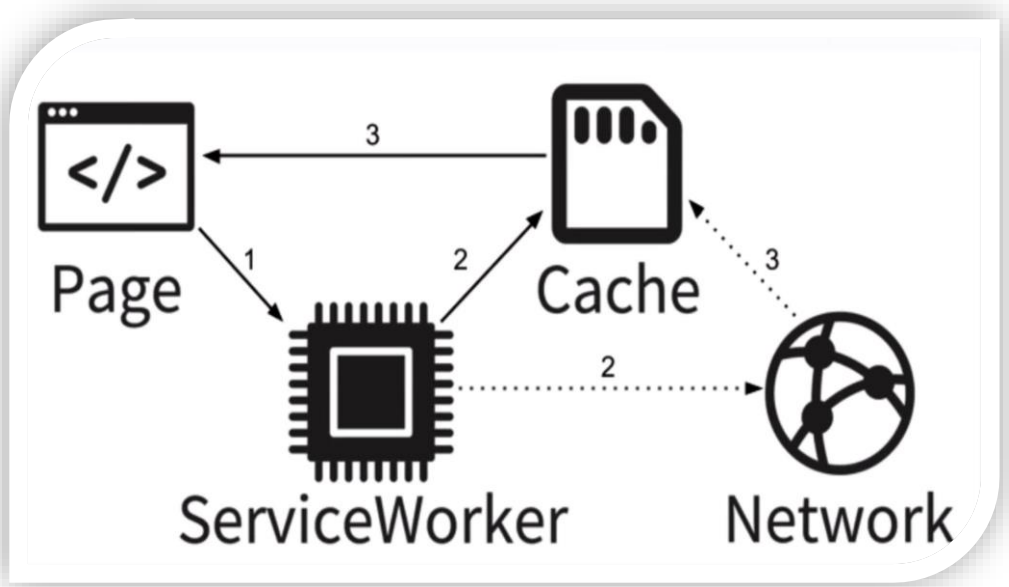
Las características o restricciones más destacadas de un *service worker* son:

- Son *JavaScript Workers*, así que no pueden acceder al DOM directamente. Como alternativa, un *service worker* puede comunicarse con las páginas que controla porque responde a mensajes enviados a través de la interfaz de *`postMessage`*; desde estas páginas se puede manipular el DOM si es necesario.

- Un *service worker* es un proxy de red programable, por lo que permite controlar la manera en que se procesan las solicitudes de red de la página web.

service worker

- Los *service workers* hacen gran uso de las promesas.
- Pueden implementar diferentes sistemas de cache.
- Se detiene cuando no está en uso y se reinicia cuando se lo necesita nuevamente, así que no se puede confiar en el estado global de los controladores `onfetch` y `onmessage` de un *service worker.*



Si se necesita hacer uso de información persistente, los *service workers* tienen acceso a la API de *IndexedDB* (parecido a *localSotarge* pero utilizando una especie de base de datos, lo que permite almacenar más información y, como está indexada, la información es fácilmente accesible).

service worker

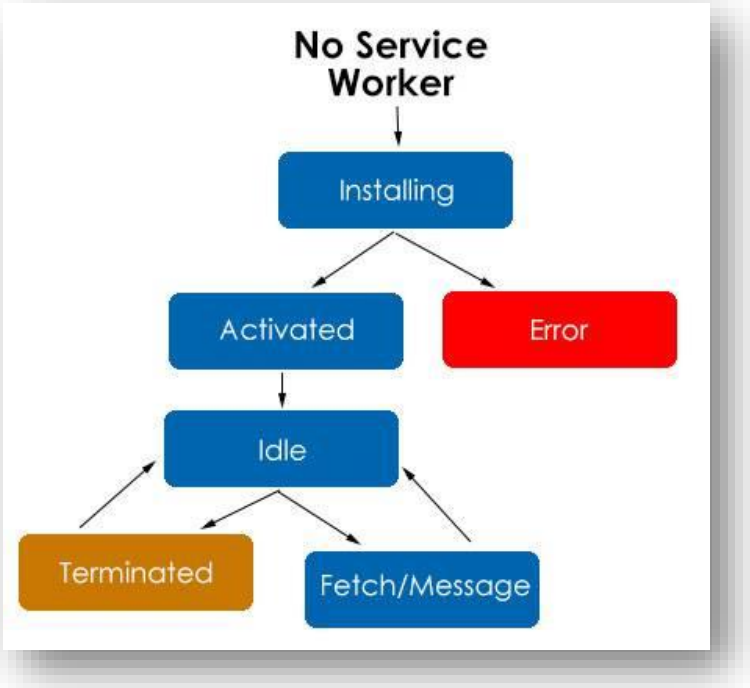
El ciclo de vida de un *service worker* es totalmente independiente de la página web y su fin es conseguir la mejor experiencia de usuario.

A continuación, se explican las diferentes etapas más importantes del ciclo de vida de un *service worker*.

Instalación. Antes de instalar un *service worker* se debe registrar el script de JS.

Una vez registrado, el navegador comienza la etapa de instalación en segundo plano.

La instalación solamente se realiza una vez por *service worker* y en esta etapa se pueden hacer configuraciones que son necesarias para el correcto funcionamiento, como, por ejemplo, almacenar o preparar una caché.



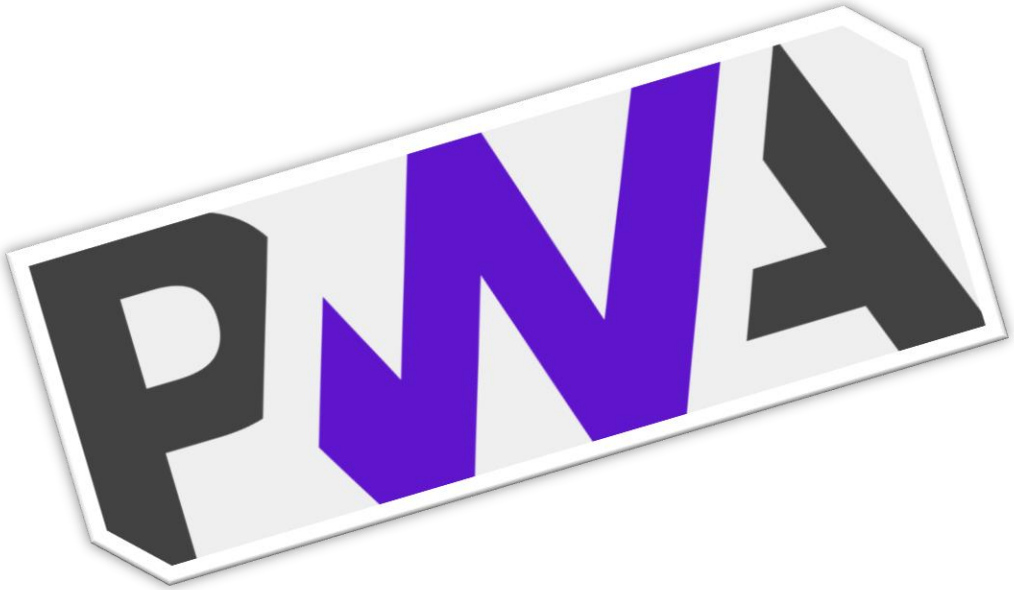
service worker

Activación. Cuando el *service worker* se instala correctamente podrá controlar el navegador y pasará al estado *Activate* o activo, manejando eventos.

Terminated y **Fetch**. Tras la activación, el *service worker* controlará todas las páginas posibles, pudiendo aparecer dos estados diferentes: *Terminated* o modo de ahorro de memoria y *Fetch*, que indica que está manejando las peticiones de red.

Los *service worker* son la tecnología que hace funcionar las *PWA* (*Progressive Web Apps*) o aplicaciones web progresivas.

Las PWA son como cualquier otra aplicación, se parecen a una app nativa para teléfonos móviles y tablets, pero hacen uso de una solución basada en la web.



Bibliografía y recursos online

- https://html.spec.whatwg.org/multipage/dnd.html#the-drag-data-store
- https://developer.mozilla.org/en-US/docs/Web/API/IntersectionObserverEntry
- https://developer.mozilla.org/es/docs/Web/API/Navigator
- https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API
- https://datatracker.ietf.org/doc/html/rfc5646
- https://www.sitepoint.com/iso-2-letter-language-codes/