

DESARROLLO WEB
ENTORNO CLIENTE

2º DAW
CFGs

6. JS: PROGRAMACIÓN ASÍNCRONA. APIs



Natalia Escrivá Núñez

IES Serra Perenxisa

n.escrivanunez@edu.gva.es

CONTENIDO

Contenido.....	3
1. CALLBACKS Y PROMISES	4
1.1. Callback	4
1.2. Callback Hell.....	6
1.3. Promise.....	7
2. APIs	12
2.1. Notification API	13
2.2. Intersection Observer API.....	15
2.3. Navigator API	16
2.4. FullScreen API.....	16
2.5. Speech Recognition API	17
2.6. Fetch API	18
3. MANEJO DE ERRORES.....	24
3.1. try() & catch()	24
4. ASYNC AWAIT	25
4.1. Async await con varias promesas.....	27
4.2. Async await con fetch()	28



1. CALLBACKS Y PROMISES

La mayoría de los lenguajes de programación son síncronos. Esto significa que, la línea 9 del código NO se ejecutará hasta que la tarea que realiza la línea 8 haya terminado.

JavaScript es un lenguaje **asíncrono**. Esto implica que una instrucción puede no haber terminado su labor cuando ya se está ejecutando la siguiente.

Vimos un ejemplo de la **asincronía** con los **eventos**:

```
console.log(1);
window.addEventListener("load", () =>{
  console.log(2);
});
console.log(3);
```

```
1
3
Live reload enabled.
2
```

Otro ejemplo claro de la necesidad de la asincronía es si **nuestro código depende de diferentes servicios**: Por ejemplo, queremos cargar la temperatura prevista para hoy, proporcionada por un servicio externo y, por otro lado, un mapa que muestre cómo llegar a la oficina, usando un servicio de mapas de Internet.

Si fuera un lenguaje síncrono, el mapa NO se cargaría hasta que la temperatura se hubiese mostrado → NO tiene sentido.

En los **lenguajes síncronos** para dar este efecto, crean **varios hilos independientes** en los que cada uno realiza una tarea.

JavaScript es un lenguaje **de un solo hilo**, pero las **operaciones sobre la red y otras de entrada/salida** (como consultas a la BBDD), se lanzan de forma **independiente**.

Pero ese efecto también tiene sus detalles a cuidar. Si, por ejemplo, quiero colorear un mapa, necesito que antes se cargue.

La **solución a la necesidad de sincronía** la teníamos con las **funciones callback**.

1.1. Callback

Son aquellas funciones que son argumento de otra función.

Hemos visto muchos ejemplos de callbacks a lo largo del curso:

```
letters.forEach((element, index) => {
  console.log(`pos=${index} letter=${element}`);
});
```



```
function action() {
  console.log("He ejecutado la función");
}
```

```
setTimeout(action, 2000);
```

```
const resultado = coches.filter(filtrarMarca).filter(filtrarAnyo)
                        .filter(filtrarPvpMin).filter(filtrarPvpMax)
                        .filter(filtrarPuertas).filter(filtrarTransmision).filter(filtrarColor);
```

Veamos algunos ejemplos nuevos:

Ejemplo 1 → Función que tras inicio de sesión, invoca la función ‘saludar’

```
function saludar(nombre){
  console.log('Hola ' + nombre);
}

function darBienvenidaUsuario(callback){
  const nombre = prompt('Por favor, indica tu nombre...');
  callback(nombre);
}

darBienvenidaUsuario(saludar);
```

Ejemplo 2 → Descarga de lista de datos y añadir otros después (a través de una callback)

Partimos de un array de países:

```
const paises = ['Francia', 'España', 'Portugal'];
```

Tenemos una función que, pasado un segundo, muestra el contenido del array:

```
function mostrarPaises(){
  setTimeout(()=>{
    paises.forEach(pais =>{
      console.log(pais)
    })
  }, 1000);
}

mostrarPaises();
```

```
Francia
España
Portugal
```

Creamos ahora la función callback que, introducirá otro país y mandará llamar a la función que los muestra:

```
function nuevoPais(pais, callback){
  setTimeout(() =>{
    paises.push(pais);
    callback();
  }, 2000);
}

nuevoPais('Italia', mostrarPaises);
```

```
Francia
España
Portugal
Italia
```



1.2. Callback Hell

Si la práctica anterior la repetimos mucho, podemos caer en lo que llaman **Callback Hell**

En el ejemplo anterior solo se añade un país pero... ¿Qué ocurre si queremos ir añadiendo más y más países?

La solución es repetir las llamadas a las funciones *callback*.

Ejemplo 1 callback hell → Creamos una estructura con varias funciones que van a ir añadiendo frutas a un array y lo van a ir mostrando.

Partimos de un array vacío:

```
const frutas = [];
```

Función *callback* que añade la fruta y llama a la función que muestra el array:

```
function nuevaFruta(fruta, callback){
  frutas.push(fruta);
  console.log(`Añadida la fruta: ${fruta}`);
  callback();
}
```

Con una sola fruta:

```
function iniciarCallbackHell(){
  setTimeout(()=>{
    nuevaFruta('Fresa', mostrarFrutas);
  }, 3000);
}

iniciarCallbackHell();
```

```
function mostrarFrutas(){
  console.log(frutas);
}
```

Añadida la fruta: Fresa

▶ ['Fresa']

Con 5 frutas:

```
function iniciarCallbackHell(){
  setTimeout(()=>{
    nuevaFruta('Fresa', mostrarFrutas);
    setTimeout(()=>{
      nuevaFruta('Kiwi', mostrarFrutas);
      setTimeout(()=>{
        nuevaFruta('Melocotón', mostrarFrutas);
        setTimeout(()=>{
          nuevaFruta('Pera', mostrarFrutas);
          setTimeout(()=>{
            nuevaFruta('Manzana', mostrarFrutas);
          }, 3000);
        }, 3000);
      }, 3000);
    }, 3000);
  }, 3000);
}

iniciarCallbackHell();
```

Añadida la fruta: Fresa

▶ ['Fresa']

Añadida la fruta: Kiwi

▶ (2) ['Fresa', 'Kiwi']

Añadida la fruta: Melocotón

▶ (3) ['Fresa', 'Kiwi', 'Melocotón']

Añadida la fruta: Pera

▶ (4) ['Fresa', 'Kiwi', 'Melocotón', 'Pera']

Añadida la fruta: Manzana

▶ (5) ['Fresa', 'Kiwi', 'Melocotón', 'Pera', 'Manzana']



Ejemplo 2 callback hell → Imaginamos un botón que al hacer click sobre él, va a cargar un mapa. Además, tras colorearlo, queremos que haya una animación sobre él.

Todo ello sin errores anteriores, perdería el sentido la traza.

El siguiente código hace que cada función reciba una variable que marca si hubo error para poder seguir con el proceso.

```

boton.addEventListener('click', ()=>{
  cargarMapa(componente, function(error){
    if (error){
      console.log('Error al cargar el mapa');
    }else{
      colorearMapa(componente, function(error){
        if (error){
          console.log('Error al colorear')
        }else{
          animarMapa(componente)
        }
      })
    }
  })
})

```

Claramente la invocación a 'animarMapa' se hace una vez se haya ejecutado 'colorearMapa' y esta se hará si 'cargarMapa' se ha resuelto con éxito.

El planteamiento es correcto pero muy farragoso.

La **solución** a estos problemas son las llamadas **Promesas**.

1.3. Promise

Estructura que permite controlar de forma más organizada las tareas asíncronas sin tener que utilizar tantas funciones callback anidadas.

Sustituye a la estructura vista en el punto anterior.

Las promesas se implementaron primero en librerías de terceros, pero la norma ES2015 las incorporó al estándar y, desde entonces, su uso se ha extendido considerablemente por la comunidad de desarrolladores.

Una promesa permite invocar una función o tarea, cuya labor requiere una ejecución asíncrona.

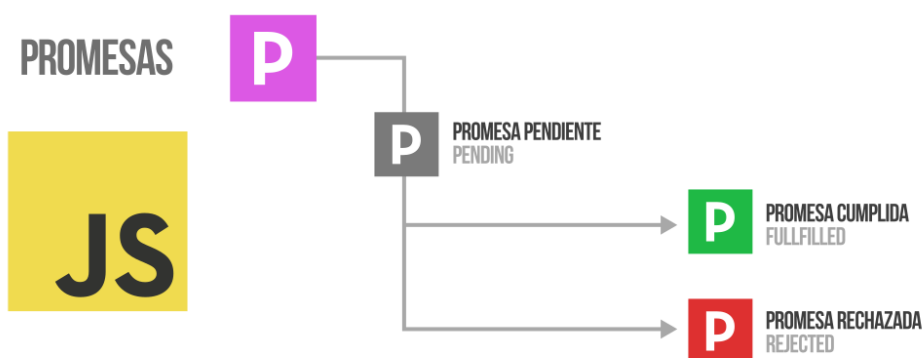
NO deja de ser una callback pero con una sintaxis más amigable y clara.



Las promesas son **objetos de tipo Promise**. Estos objetos relacionan la labor **asíncrona** con las **acciones a tomar en caso de éxito o fracaso**. Para ello proporcionan varios métodos y una función de construcción de objetos **Promise**.

Una promesa puede tener uno de estos **estados**:

- Pendiente de finalizar (**pending**): En proceso de finalización.
- Cumplida (**resolved** o **fulfilled**): Si la tarea relacionada con la promesa se ha finalizado con éxito. Procesaremos la información con el **método.then()**
- Rechazada (**rejected**): Si la tarea no finaliza con éxito. Procesaremos el error con el **método.catch()**



1.3.1. CREAR UNA PROMESA

Las promesas se construyen como objetos de tipo Promise.

Tienen dos parámetros que serán dos funciones callback: La primera es la de **resolver** y se invoca cuando se ha verificado que la operación ha finalizado de forma correcta y la **segunda** es la de **rechazar** y se invoca cuando se ha determinado que el proceso NO se ha ejecutado correctamente.

Ambas funciones reciben parámetros. Para la **función de rechazo** el parámetro suele ser un objeto de error, ya que permite la gestión de errores de forma más conveniente.

La creación del objeto de promesa implica lanzar la tarea en segundo plano.



Ejemplo 1 → Vemos los tres estados: fulfilled, rejected y pending

El resultado es OK → resolve: La conexión al servidor ha sido exitosa y hemos podido acceder y aplicar descuentos.

```
const aplicarDescuento = new Promise((resolve, reject) =>{
  const descuento = true;
  if (descuento) {
    resolve('Descuento aplicado');
  }else{
    reject('NO se pudo aplicar el descuento');
  }
})
console.log(aplicarDescuento);
```

```
► Promise {<fulfilled>: 'Descuento aplicado'}
```

El resultado es NO OK → rejected: La conexión al servidor ha fallado y no hemos podido realizar la conexión.

```
const aplicarDescuento = new Promise((resolve, reject) =>{
  const descuento = false;
  if (descuento) {
    resolve('Descuento aplicado');
  }else{
    reject('NO se pudo aplicar el descuento');
  }
})
console.log(aplicarDescuento);
```

```
► Promise {<rejected>: 'NO se pudo aplicar el descuento'}
```

```
✖ ► Uncaught (in promise) NO se pudo aplicar el descuento
```

El resultado es indefinido → El resultado no se ha definido, ni resolve ni reject. NO hay resultado porque no se ha cumplido pero tampoco ha sido rechazado.

```
const aplicarDescuento = new Promise((resolve, reject) =>{
  const descuento = false;
  if (descuento) {
    //resolve('Descuento aplicado');
  }else{
    // reject('NO se pudo aplicar el descuento');
  }
})
console.log(aplicarDescuento);
```

```
▼ Promise {<pending>} ⓘ
  ► [[Prototype]]: Promise
  [[PromiseState]]: "pending"
  [[PromiseResult]]: undefined
```



1.3.2. MÉTODOS THEN() Y CATCH()

Usaremos estos dos métodos para recoger y poder usar los resultados de las promesas.

Seguimos con el ejemplo anterior...

```
aplicarDescuento
  .then(resultado =>{
    console.log(`Resultado de la promesa: ${resultado}`);
  })
  .catch(error =>{
    console.log(`Resultado de la promesa: ${error}`);
  })
```

Si descuento = true...

```
Resultado de la promesa: Descuento aplicado
```

Si descuento = false...

```
Resultado de la promesa: NO se pudo aplicar el descuento
```

Podemos usar otras sintaxis:

```
aplicarDescuento.then(resultado => console.log(resultado)).catch(error => console.log(error));
```

```
aplicarDescuento.then(resultado => console.log(resultado), error => console.log(error));
```

También podemos llamar a funciones dentro de los then() o catch()...

```
aplicarDescuento
  .then(resultado => descuento(resultado))
  .catch(error => console.log(`Resultado de la promesa: ${error}`));

function descuento(mensaje){
  console.log(mensaje);
}
```



1.3.3. PASAR DE CALLBACK A PROMISES

Vamos a simular la misma acción con el array de frutas del apartado de Callback Hell con Promesas.

1. Partimos del array de frutas vacío:

```
const frutas = [];
```

2. La función callback que añadía la fruta al array y llamaba a la función que mostraba el array la sustituimos por la Promise:

```
function nuevaFruta(fruta, callback){
  frutas.push(fruta);
  console.log(`Añadida la fruta: ${fruta}`);
  callback();
}

const nuevaFruta = fruta => new Promise (resolve =>{
  setTimeout(()=>{
    frutas.push(fruta);
    resolve(`Resultado del Resolve:`);
  }, 3000);
})
```

No usamos reject porque es un código muy sencillo y no va a salir mal.

3. Llamamos al Promise y en “then()” llamamos a la función que muestra el array y volvemos a llamar al Promise con otra fruta y así sucesivamente...

```
function mostrarFrutas(){
  console.log(frutas);
}
```

```
function iniciarCallbackHell(){
  setTimeout(()=>{
    nuevaFruta('Fresa', mostrarFrutas);
    setTimeout(()=>{
      nuevaFruta('Kiwi', mostrarFrutas);
      setTimeout(()=>{
        nuevaFruta('Melocotón', mostrarFrutas);
        setTimeout(()=>{
          nuevaFruta('Pera', mostrarFrutas);
          setTimeout(()=>{
            nuevaFruta('Manzana', mostrarFrutas);
          }, 3000);
        }, 3000);
      }, 3000);
    }, 3000);
  }, 3000);
}

iniciarCallbackHell();
```

```
nuevaFruta('Manzana')
  .then(resultado => {
    console.log(resultado);
    mostrarFrutas();
    return nuevaFruta('Fresa');
  })
  .then(resultado => {
    console.log(resultado);
    mostrarFrutas();
    return nuevaFruta('Pera');
  })
  .then(resultado => {
    console.log(resultado);
    mostrarFrutas();
    return nuevaFruta('Kiwi');
  })
  .then(resultado => {
    console.log(resultado);
    mostrarFrutas();
    return nuevaFruta('Naranja');
  })
```



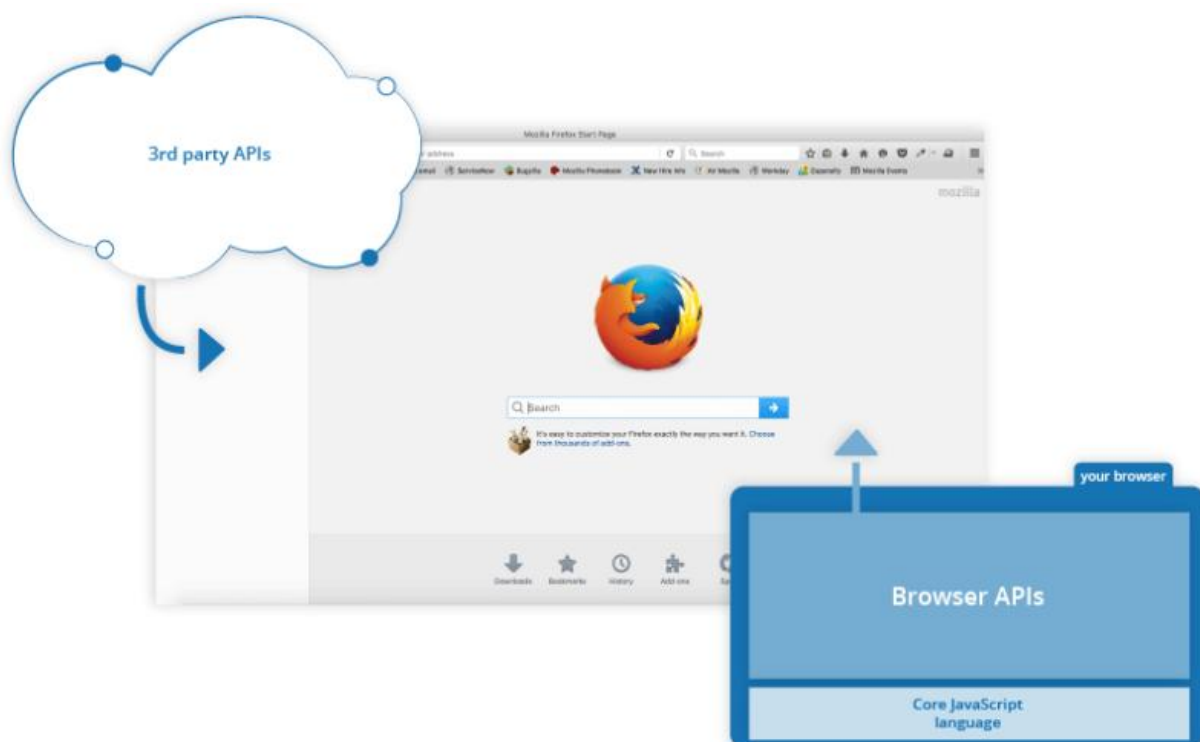
2. APIS

Las Interfaces de Programación de Aplicaciones (APIs) son construcciones disponibles que abstraen el código más complejo para proveer una sintaxis sencilla de usar.

Responden a una **interfaz de procesamiento de aplicaciones entre un servidor web y un navegador web**.

JavaScript tiene muchas APIs disponibles. Están divididas en dos categorías:

- **APIs de navegador:** integradas en el propio navegador web, por lo que pueden exponer datos del navegador y del entorno informático de los ficheros ejecutados. Por ejemplo, la API de Geolocalización.
- **APIs de terceros:** no incluidas en el navegador, por lo que es necesario obtener el código e información desde algún lugar de la Web. Por ejemplo, la API de X (antes Twitter), que permite incluir en tu página web tus publicaciones (antes tweets).



Hay muchas APIs, y mucha documentación sobre ellas, vamos a ver un pequeño ejemplo de algunas que pueden ser útiles.

Las APIs utilizan **Promises** por lo que usaremos su sintaxis.



2.1. Notification API

Es una **API nativa (de navegador)** de JavaScript.

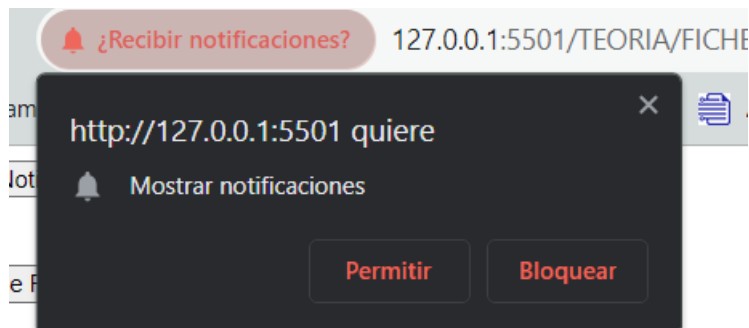
Es una API que nos va a permitir enviar notificaciones al usuario.

Debemos saber que para enviar notificaciones a los usuarios, primero debemos obtener su permiso.

Vamos a utilizar un botón del HTML para asociarle un evento y cuando cliclemos sobre él, llamará a **Notification API**.

1. Pedimos permiso al usuario llamando a Notification API → Dependiendo del comportamiento del usuario el mensaje es uno u otro.

```
const notificarBtn = document.querySelector('#notificar');
notificarBtn.addEventListener('click', () =>{
  //usamos Notification API
  Notification
    .requestPermission()
    .then(resultado =>{
      console.log(`El resultado es: ${resultado}`)
    })
})
```



El resultado es: granted

El resultado es: denied

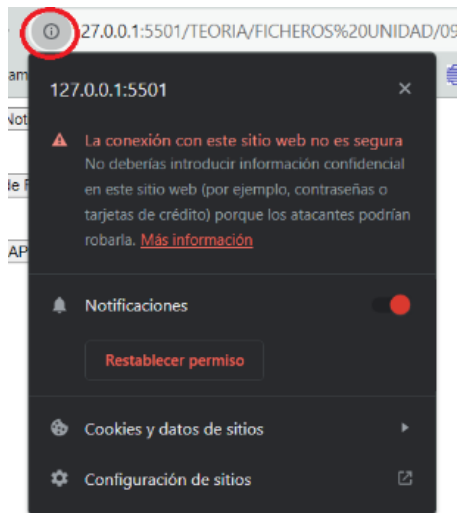
El resultado es: default

Tenemos tres opciones de respuesta:

- **Granted:** el usuario ha concedido el permiso explícitamente al origen actual para mostrar notificaciones del sistema.
- **Denied:** el usuario ha denegado el permiso explícitamente al origen actual para mostrar notificaciones del sistema.
- **Default:** La decisión del usuario es desconocida; en este caso la aplicación actuará como si el permiso fuese **denied**.

Ahora esa configuración queda registrada en los ajustes del navegador, por lo que no volverá a preguntar.

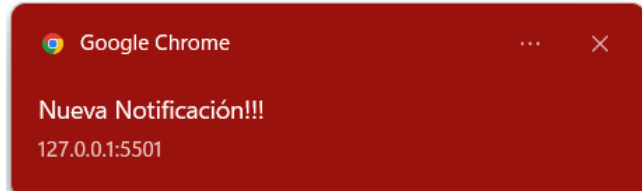




****Para restablecer los permisos, cliclamos sobre el icono de info**

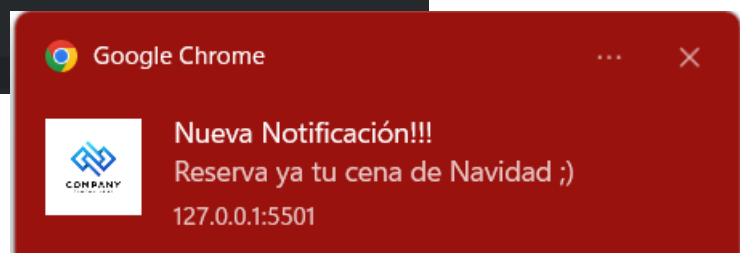
2. Creamos una notificación básica, que aparecerá cuando cliquemos sobre el botón del html “Ver Notificación”:

```
const verNotificacionBtn = document.querySelector('#verNotificacion');
verNotificacionBtn.addEventListener('click', ()=>{
  if (Notification.permission === 'granted'){
    new Notification('Nueva Notificación!!!');
  }
});
```



3. Mejoramos la notificación con nuestro logo, el mensaje concreto y acceso a la web cuando cliquen sobre la url:

```
const verNotificacionBtn = document.querySelector('#verNotificacion');
verNotificacionBtn.addEventListener('click', ()=>{
  if (Notification.permission === 'granted'){
    const notificacion = new Notification('Nueva Notificación!!!', {
      icon: 'img/ccj.jpg',
      body: 'Reserva ya tu cena de Navidad ;)'
    });
    notificacion.addEventListener('click', ()=>{
      window.open('http://www.google.com')
    });
  }
});
```



2.2. Intersection Observer API

Esta API nos permite identificar cuando un elemento está visible. Es nativa de JS (*API de HTML DOM*)

Vimos algo parecido en los eventos con *getBoundingClientRect()*.

Presentamos Mivaje.com Plus



```

window.addEventListener('scroll', () => {
  const premium = document.querySelector('.premium');
  const ubicacion = premium.getBoundingClientRect();

  if (ubicacion.top < 750 && ubicacion.top > -500){
    console.log('El elemento ya está visible');
  } else {
    console.log('Aún no, da más scroll...');
  }
});

```

```

DOMRect {x: 107.0666732788086, y: 199.61668395996094, width:
48.0000305175781, top: 199.61668395996094, ...}
  bottom: 647.6167144775391
  height: 448.0000305175781
  left: 107.0666732788086
  right: 1207.0666732788086
  top: 199.61668395996094
  width: 1100
  x: 107.0666732788086
  y: 199.61668395996094
  ▶ [[Prototype]]: DOMRect
El elemento ya está visible

```

Con *IntersectionObserver* lo que hacemos “observar” algún elemento y así poder realizar una acción concreta cuando esté en la pantalla.

Con la API es mucho más sencillo que con el método visto anteriormente, ya que no necesitamos ocuparnos de la ubicación del elemento.

```

document.addEventListener('DOMContentLoaded', () =>{
  //creamos el IntersectionObserver
  const observer = new IntersectionObserver(entries =>{
    if(entries[0].isIntersecting){
      console.log('Ya está visible')
    }else{
      console.log('No está visible');
    }
  });

  //indicamos qué elemento va a ser observado
  observer.observe(document.querySelector('.premium'));
});

```

Esto es muy utilizado para hacer *Lazy loading*, es decir, no recargar elementos hasta que no vayan a ser utilizados o un *scroll infinito*.



2.3. Navigator API

Hacemos referencia a la **API Web**. La propiedad de solo lectura **window.navigator** es una referencia al objeto Navigator, que puede ser utilizado para obtener información sobre la aplicación que se está ejecutando.

Nos puede ser muy útil cuando el usuario intenta realizar una acción en Internet (completar un pedido, mandar una publicación, solicitar un servicio...) y debemos asegurarnos de que **tiene conexión**.

Vamos a ver cómo su sintaxis:

```

window.addEventListener('online', actualizarEstado);
window.addEventListener('offline', actualizarEstado);

function actualizarEstado(){
    if (navigator.onLine){
        console.log('Tienes conexión a Internet')
    }else{
        console.log('No estás conectado...')
    }
}

```

2.4. FullScreen API

Hemos visto como trabajar con la **API de HTML DOM**.

Una funcionalidad que podemos utilizar es **ejecutar código en pantalla completa** (como hacemos cuando maximizamos un video de Youtube).

Lo que debemos tener en cuenta es que **debemos indicar qué elemento del HTML queremos que se ejecute en pantalla completa** (una foto, un video...). En este caso, seleccionamos todo el HTML con **document.documentElement**.

```

const abrirBtn = document.querySelector('#abrir-pantalla-completa');
const salirBtn = document.querySelector('#salir-pantalla-completa');

abrirBtn.addEventListener('click', pantallaCompleta);
salirBtn.addEventListener('click', cerrarPantallaCompleta);

function pantallaCompleta(){
    document.documentElement.requestFullscreen();
}

function cerrarPantallaCompleta(){
    document.exitFullscreen();//esc
}

```



2.5. Speech Recognition API

Con esta API podremos detectar lo que el usuario habla y trasladarlo a la página web. Es lo que se llaman los “asistentes por voz”.

Debemos tener en cuenta que debemos dar permisos al micrófono.

No todos los navegadores soportan esta API (Firefox de momento no).

En los datos que recoge podemos ver distintos valores:

- **Confidence:** valor entre 0 y 1 que nos indica el grado de legibilidad del mensaje detectado.
- **Transcript:** el audio que se ha detectado.

El proceso tiene varias partes....

1. Seleccionamos los elementos del HTML relacionados.

```
<button type="button" id="microfono">
  Speech Recognition API
</button>

<div id="salida" class="ocultar"></div>
```

```
const salida = document.querySelector('#salida');
const microfono = document.querySelector('#microfono');
```

2. Al botón le asignamos el evento.

```
microfono.addEventListener('click', ejecutarSeechAPI);
```

3. Ya en la función:

- 3.1. Creamos la variable para usarlo en el navegador y un nuevo objeto.

```
const SpeechRecognition = webkitSpeechRecognition;
const recognition = new SpeechRecognition();
```

- 3.2. Iniciamos *Speech API*

```
recognition.start();
```

- 3.3. Asignamos el evento y la función que recogerá el audio

```
recognition.addEventListener('start', ()=>{
  salida.classList.add('mostrar');
  salida.textContent = 'Escuchando...';
})
```



3.4. Asignamos el evento y función de parada cuando no detecta sonido

```
recognition.addListener('speechend', ()=>{
  setTimeout(()=>{
    salida.textContent = 'Grabación finalizada!';
  },2000);
  recognition.stop();
})
```

3.5. Asignamos el evento y función de la transcripción de los resultados

```
recognition.addListener('result', transcribirAudio);

function transcribirAudio(e){
  salida.textContent = '';
  const {confidence, transcript} = e.results[0][0];
  console.log(e.results)
  const speech = document.createElement('p');
  speech.innerHTML = `Grabado: ${transcript}`;

  const seguridad = document.createElement('p');
  seguridad.innerHTML = `Seguridad: ${parseInt(confidence * 100)}%`;

  salida.appendChild(speech);
  salida.appendChild(seguridad);
}
```

Speech Recognition API
Escuchando...

Speech Recognition API
Grabado: esto es una prueba
Seguridad: 95%

Speech Recognition API
Grabación finalizada!

2.6. Fetch API

Es una API que nos va a servir para obtener datos de otros servidores o APIs. Es nativa.

A través de esta API podremos enviar y recibir datos.

Es una API que solo puede leer texto plano o JSON. Por ejemplo XML no lo soporta. Antes de esta API se utilizaba AJAX.

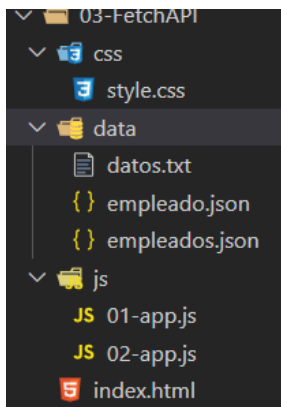
Para su ejecución utiliza Promises que tiene implementados (no necesitamos hacer la promesa con resolve y reject).



La llamada a *Fetch API* genera un *objeto de tipo Response*. La información más importante es:

- **Status:** código que informa del estado de la solicitud
- **Status text:** mensaje de estado más legible
- **url:** dirección de donde leemos o escribimos datos.
- **Type:** tipo de transacción

Los ficheros que disponemos para esta parte de la unidad son:



Carpetas css, js y fichero html

Carpeta 'data' que incluye un fichero de texto y dos ficheros json.

Vamos a ver cómo accedemos a estos ficheros a través de fetch API.

2.6.1. CONSULTAR UN TXT

Una vez seleccionado el elemento del HTML (en este caso, un botón) y asignado el evento, en la función debemos llamar a la API con la url. Iremos indicando con then() y catch() las acciones a realizar:

```
const url = 'data/datos.txt';

fetch(url)
//analizamos respuesta de la promesa, si conecta...
.then(respuesta => {
  console.log(respuesta);
  console.log(respuesta.status);
  console.log(respuesta.statusText);
  console.log(respuesta.url);
  console.log(respuesta.type);

  return respuesta.text();
})
//si lo anterior es resolve...
.then(datos => {
  console.log(datos);
})
//si encontramos errores...
.catch(error => {
  console.log(error);
})
```

Informacion desde un archivo .txt

```
200
OK
http://127.0.0.1:5501/TEORIA/FICHEROS%20UNIDAD/10-FetchAPI/data/datos.txt
basic
404
Not Found
http://127.0.0.1:5501/TEORIA/03-FetchAPI/data/datos.txt
basic
```



2.6.2. CONSULTAR UN JSON COMO OBJETO

JSON se considera una tecnología de intercambio de datos. Permite separar los lenguajes del emisor y receptor de los datos.

JavaScript **NO** puede consultar directamente una base de datos (a excepción de Firebase, que crea una capa de acceso) por lo que, independientemente de qué lenguaje estemos utilizando en el backend, con JSON podremos acceder a los datos que necesitemos.

Un JSON es bastante parecido a un objeto. La diferencia es que los elementos (clave: valor) se encuentran entre comillas, sea cual sea su tipo de contenido.

Partimos de un fichero con un objeto JSON (empleado.json):

```
{
  "id" : 1,
  "nombre" : "Roberto López Buhadilla",
  "empresa" : "Transportes Buhadilla",
  "trabajo" : "Gerente"
}
```

Vamos a seleccionar el elemento del html que tendrá el evento (botón)

```
const cargarJSONBtn = document.querySelector('#cargarJSON');
cargarJSONBtn.addEventListener('click', obtenerInfo);
```

Ya en la función “obtenerInfo”...

Lanzamos fetch:

```
function obtenerInfo(){
  fetch('data/empleado.json')
```

Analizamos la respuesta del Promise...

```
.then(respuesta =>{
  console.log(respuesta);
  return respuesta.json();
})
//mostramos los datos en formato objeto...
.then(resultado => mostrarHTML(resultado));
```

Si clicamos sobre el botón “Cargar JSON - Objeto”...

```
EMPLEADO: ROBERTO LÓPEZ BUHADILLA
ID EMPLEADO: 1
EMPRESA: TRANSPORTES BUHADILLA
FUNCIÓN: GERENTE
```



2.6.3. CONSULTAR UN JSON COMO ARRAY

```
[
  {
    "id" : 1,
    "nombre" : "Roberto López Buhadilla",
    "empresa" : "Desarrollo en un click",
    "trabajo" : "Diseño"
  },
  {
    "id" : 2,
    "nombre" : "Alejandra Martínez López",
    "empresa" : "Desarrollo en un click",
    "trabajo" : "Desarrollo frontend"
  },
  {
    "id" : 3,
    "nombre" : "Pedro Masiá Gómez",
    "empresa" : "Desarrollo en un click",
    "trabajo" : "Desarrollo backend"
  }
]
```

Partimos de un fichero /empleados.json con un array de JSON:

Vamos a seleccionar el elemento del html que tendrá el evento (botón)

```
const cargarJSONArrayBtn = document.querySelector('#cargarJSONArray');
cargarJSONArrayBtn.addEventListener('click', obtenerInfoEmpleados);
```

Ya en la función “obtenerInfoEmpleados”...

Lanzamos fetch y analizamos la respuesta del Promise...

```
function obtenerInfoEmpleados(){
  const url = 'data/empleados.json';

  fetch(url)
    .then(respuesta => respuesta.json())
    .then(respuesta => mostrarHTMLArray(respuesta));
}
```

```
EMPLEADO: ROBERTO LÓPEZ BUHADILLA
ID EMPLEADO: 1
EMPRESA: DESARROLLO EN UN CLICK
FUNCIÓN: DISEÑO
*****
EMPLEADO: ALEJANDRA MARTÍNEZ LÓPEZ
ID EMPLEADO: 2
EMPRESA: DESARROLLO EN UN CLICK
FUNCIÓN: DESARROLLO FRONTEND
*****
EMPLEADO: PEDRO MASIÁ GÓMEZ
ID EMPLEADO: 3
EMPRESA: DESARROLLO EN UN CLICK
FUNCIÓN: DESARROLLO BACKEND
*****
```

Si cliclamos sobre el botón “Cargar JSON - Array”...



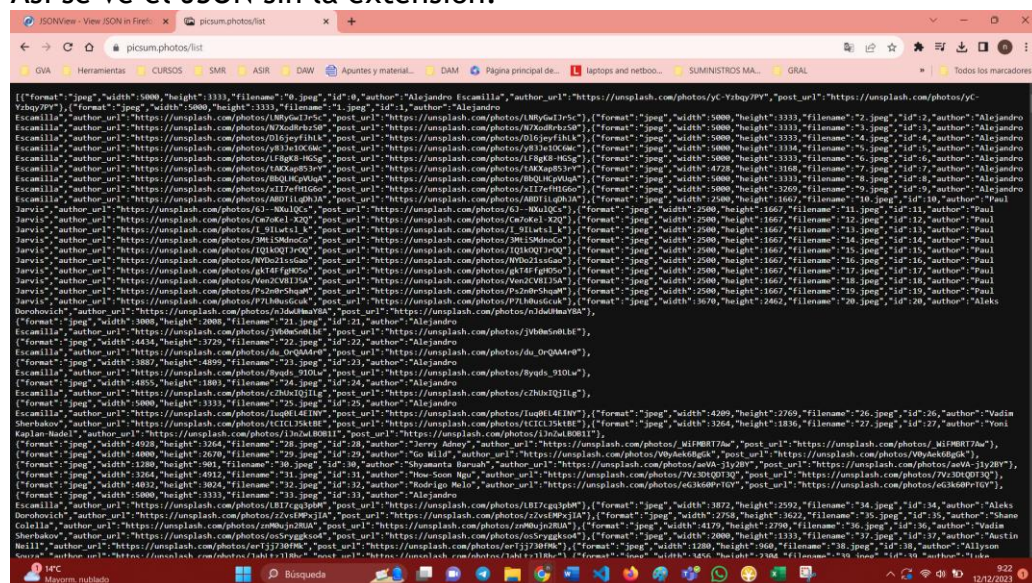
2.6.4. CONSULTAR E IMPRIMIR LOS RESULTADOS DE UNA API

Vamos a consumir datos de una API (picsum.photos/list) de una web de fotos gratuitas *unsplash*.

Para poder ver mejor los datos (JSON) que devuelve la API, podremos usar la extensión para el navegador JSONVIEW (jsonview.com)



Así se ve el JSON sin la extensión:



Así se ve el JSON con la extensión:



El procedimiento es el mismo que con los ejemplos anteriores. En este caso sí podemos tener en cuenta que falle la conexión a Internet y podemos añadir `catch()` al Promise.

```
const cargarAPIBtn = document.querySelector('#cargarAPI');
cargarAPIBtn.addEventListener('click', obtenerDatosAPI);

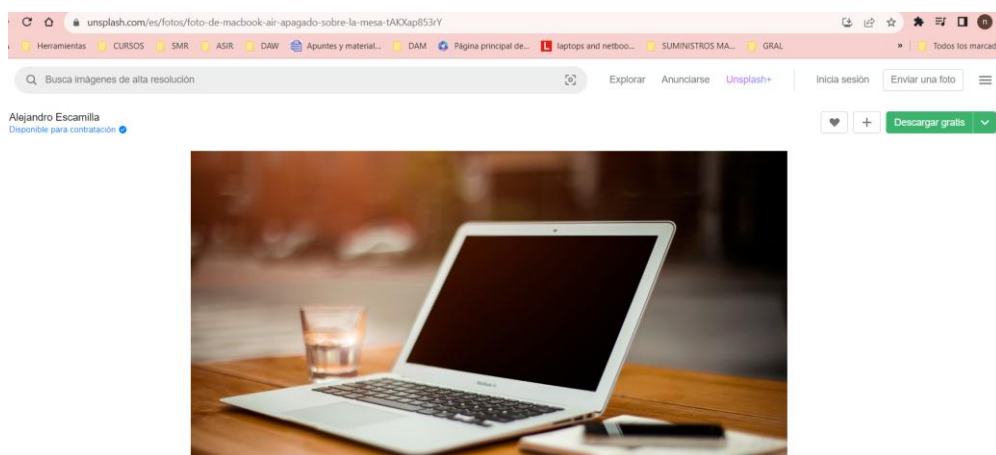
function obtenerDatosAPI(){
  const url = 'https://picsum.photos/list';

  fetch(url)
    .then(respuesta => respuesta.json())
    .catch(error => document.write(error))
    .then(resultados => mostrarHTMLDatosAPI(resultados));
}
```

Si cliclamos sobre el botón “Cargar API”...



Vemos los enlaces en “Ver Imagen” que nos llevarán a la página concreta de la foto en unsplash.com



3. MANEJO DE ERRORES

Una de las tareas más importantes a la hora de programar es la de solucionar los errores que van apareciendo.

Los profesionales del mundo del desarrollo invierten una gran parte de su tiempo solventando errores detectados en el programa.

Lo malo es que hay veces que los errores se detectan mucho después porque ocurren circunstancias en el uso de la aplicación que no se habían tenido en cuenta.

Los errores en un programa pueden ser:

- **Errores de sintaxis:** Hablamos de errores que **se producen al escribir código por parte del programador**. Ejemplos como expresiones incorrectas, cierre de llaves olvidados, palabras clave mal escritas, etc. Estos errores son sencillos de detectar porque bien se convierten en errores escritura (cuando el propio entorno de desarrollo lo marca) o de ejecución (cuando se interpreta y ejecuta el código).
- **Errores lógicos:** Se deben a una **mala lógica al desarrollar la aplicación**. La sintaxis es correcta pero el programa no hace lo que esperamos. No hay herramienta que nos avise de eso. Es importante que se detecte en fase de producción, testeando, antes de que algún usuario lo detecte.
- **Errores del sistema:** **Provocados por circunstancias que están fuera del control del programador:** fallo de conexión de red, caída de un servicio que estamos utilizando. No podemos controlar esos fallos, pero sí intentar minimizar los daños.
- **Errores de usuario:** Provocados por **acciones inesperadas que realiza el usuario y que causan un error en el tiempo de ejecución**. En realidad, son errores lógicos que ocurren por no prever esas actuaciones (pedir un número y recibir un texto, por ejemplo).

3.1. try() & catch()

Por ser un lenguaje interpretado, JavaScript, cuando encuentra un error, no sigue ejecutando el código.

Podemos gestionar las excepciones a través de la estructura try ... catch. Esto implica que, intentaremos ejecutar una pieza de código y, si no se puede, obtendremos una excepción pero el código no deja de funcionar.



```
try{
  alert("hola");//se ejecuta
  mifunction();//da error
  alert("adiós");//no se ejecuta
}
catch (error){
  console.log(error);
}
alert(1+1); //se ejecuta
```

No todo nuestro código debe ir en un try... catch. Lo usaremos en partes críticas de la aplicación (conexión a la base de datos, consulta de API, autenticar un usuario....).

Serán acciones que, aunque fallen, nuestra aplicación pueda seguir funcionando.

****Ejemplos ampliados en los ficheros de la unidad.**

4. ASYNC AWAIT

Se introducen en el estándar ES2017 y es una alternativa a los *Promises* y funciona sobre ellos.

Simulamos una descarga de clientes de la base de datos (entendemos una conexión que dura 3 segundos).

Para ello, creamos una función que devuelve una promesa.

```
function descargarClientes(){
  return new Promise((resolve, reject) => {
    const error = false;

    setTimeout(() => {
      if(!error){
        resolve('El listado de Clientes se descargo correctamente');
      }else{
        reject('Error en la conexión');
      }
    }, 3000);
  })
}
```

Vamos a utilizar *async/await con Function Declaration*

Para ello pondremos la palabra reservada *async* a la función “padre”, es decir, aquella sobre la que se ejecuta el Promise.

Pondremos *await* en la parte que espera que se ejecute el Promise. Lo que hará await es **detener la ejecución hasta que se resuelva**. El código que haya



debajo del `await` estará bloqueado y no se ejecutará hasta que el Promise quede resuelto.

```

async function ejecutar(){
  try{
    console.log(1+1);
    const respuesta = await descargarClientes();

    console.log(2+2);
    console.log(respuesta);
  }catch{
    console.log(error);
  }
}

ejecutar();

```

```

2
4
El listado de Clientes se descargo correctamente

```

Lo que antes teníamos como respuesta de la promesa con `.then`, ahora debemos declararlo a través de una constante.

En este caso, se ejecuta el primer `console.log` y, pasados 3 segundos (del Promise de la función), se ejecutará el segundo `console.log` y la respuesta.

Vamos a utilizar `async/await` con *Function Expression*.

Utilizamos la misma función de `descargarClientes` (renombrada)

```

22 const ejecutar2 = async () =>{
23   try{
24     console.log(1+1);
25     //la respuesta del resolve que tendríamos en el .then de Promise,
26     //en async await se asigna a una nueva variable
27     const respuesta = await descargarClientes2();
28
29     console.log(respuesta);
30     console.log(2+2); //No se ejecutará hasta que el Promise se resuelva.
31   }catch(error){
32     console.log(error);
33   }
34 }
35

```

Si no hay error:

```

2                                02-app.js:24
El listado de Clientes se descargo correctamente 02-app.js:29
4                                02-app.js:30

```

Si hay error:

```

2                                02-app.js:24
Error en la conexión                02-app.js:33

```



4.1. Async await con varias promesas

Puede que tengamos un escenario en el que **unas promesas deban ejecutarse cuando se hayan resuelto otras**.

Por otro lado, puede ser también que hagamos distintas peticiones independientes entre ellas, es decir, que la ejecución de una NO dependa de otra.

Suponemos dos tareas: debemos descargar un listado de clientes (suponemos que la conexión tarda 5 segundos) y otro de pedidos (suponemos que la conexión tarda 3 segundos).

```
function descargarNuevosClientes(){
  return new Promise(resolve => {
    console.log('Descargando clientes....');

    setTimeout(() => {
      resolve('Clientes descargados');
    }, 5000);
  });
}
```

```
function descargarNuevosPedidos(){
  return new Promise(resolve => {
    console.log('Descargando pedidos....');

    setTimeout(() => {
      resolve('Pedidos descargados');
    }, 3000);
  });
}
```

Ejemplo 1 → Una petición depende de la otra

```
const app = async () =>{
  try{
    const clientes = await descargarNuevosClientes();
    console.log(clientes);
    const pedidos = await descargarNuevosPedidos();
    console.log(pedidos);
  }catch(error){
    console.log(error);
  }
}

app();
```

Podemos ver que primero se ejecuta una, y, pasados los 5 segundos, se ejecuta la segunda.

```
Descargando clientes....
Clientes descargados
Descargando pedidos....
Pedidos descargados
```



Ejemplo 2 → Una petición NO depende de la otra

```

42 const app2 = async () =>{
43   try{
44     const respuesta = await Promise.all([descargarNuevosClientes(), descargarNuevosPedidos()]);
45     console.log(respuesta);
46     console.log(respuesta[0]);
47     console.log(respuesta[1]);
48   }catch(error){
49     console.log(error);
50   }
51 }
52 app2();

```

Descargando clientes....	03-app.js:7
Descargando pedidos....	03-app.js:17
▶ (2) ['Clientes descargados', 'Pedidos descargados']	03-app.js:45
Clientes descargados	03-app.js:46
Pedidos descargados	03-app.js:47

En este caso vemos que se ejecutan de forma simultánea.

Con Promise.all, le pasamos como argumento un array con las promesas que queremos que se ejecuten.

4.2. Async await con fetch()

Vamos a ver la diferencia de usar Promises a la estructura Async await con fetch API.

Como elementos comunes a los dos métodos, la url y el evento de carga de la página...

```

const url= 'https://picsum.photos/v2/list'

document.addEventListener('DOMContentLoaded', obtenerDatos);

```

```

//Con Promises
function obtenerDatos(){
  fetch(url)
    .then(respuesta => respuesta.json())
    .then(resultado => console.log(resultado))
    .catch(error => console.log(error));
}

```

```

//Con Async await
async function obtenerDatos(){
  try{
    const respuesta = await fetch(url);
    const resultado = await respuesta.json();
    console.log(resultado);
  }catch(error){
    console.log(error);
  }
}

```

