Unidad 9
Programación Asíncrona

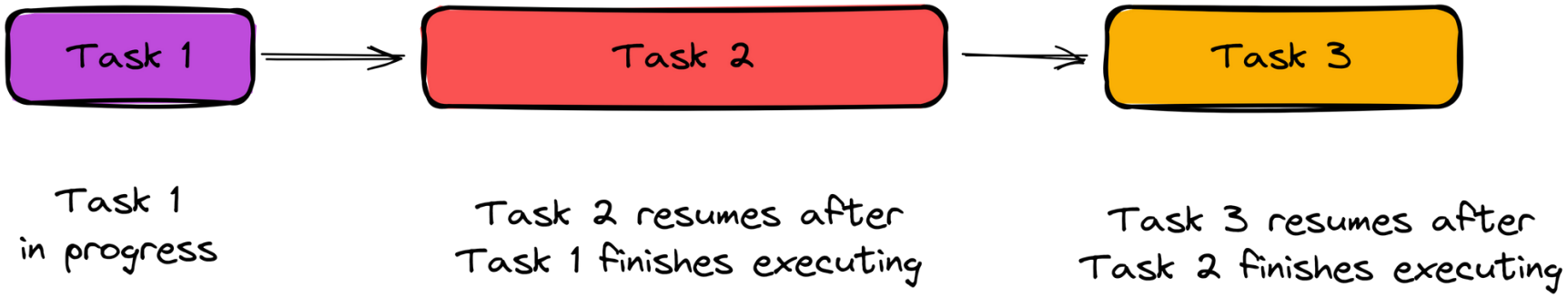
CESUR
Tu Centro Oficial de FP

1. Programación síncrona y asíncrona

PROGRAMACIÓN SÍNCRONA – técnica que se utiliza para que las computadoras realicen tareas paso a paso, en el orden en que se les dan las instrucciones.

La programación síncrona es un problema sobre todo cuando se trata de tareas que requieren una cantidad significativa de tiempo para completarse.

Por ejemplo, si tenemos un programa síncrono que realiza una tarea que requiere esperar una respuesta del servidor remoto. Mientras que el servidor no responda el programa quedará a la espera de una respuesta (bloqueo) esto puede hacer que la aplicación no responda o aparezca como congelada para el usuario.



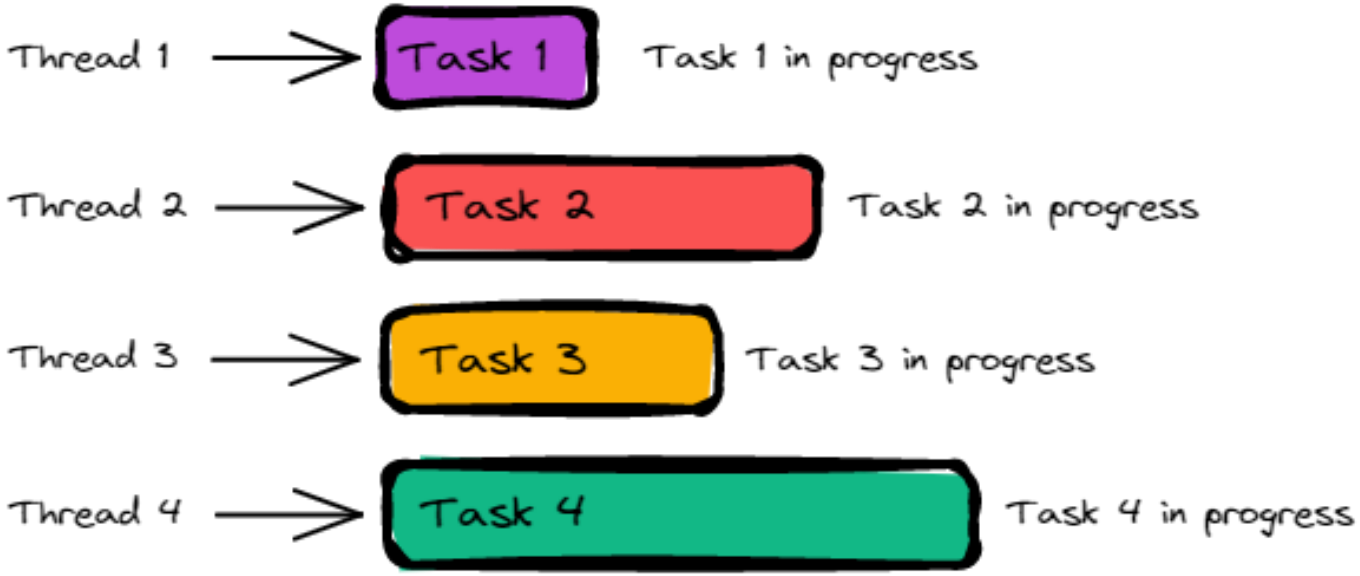
1. Programación síncrona y asíncrona

ASÍNCRONO – suceso o proceso que se produce de forma independiente al tiempo de procesamiento.

La comunicación asíncrona es una de las cualidades que confiere a JS su máxima potencia.

PROGRAMACIÓN ASÍNCRONA - permite al programa trabajar en distintas tareas simultáneamente sin tener que esperar a completar una tarea para pasar a la siguiente. Esto permite que un programa sea capaz de hacer más cosas en menos tiempo.

En el ejemplo anterior, si fuera programación asíncrona, el programa podría seguir ejecutando la siguiente línea de código en lugar de bloquearse, es decir, continúa ejecutando instrucciones mientras espera que se complete la respuesta del servidor.



1. Programación síncrona y asíncrona

Ejemplo: PROGRAMACIÓN ASÍNCRONA CON setTimeout

```
console.log("Inicio del guión");

setTimeout(function() {
  console.log("Primer tiempo muerto completado");
}, 2000);

console.log("Fin del guión");
```

```
Inicio del guión
Fin del guión
Primer tiempo muerto completado
```

En este ejemplo, el método **setTimeout** ejecutará una función después de 2000 ms (2 segundos).

La función que se le pasa a **setTimeout** se ejecutará de forma asíncrona, de tal forma que el programa seguirá ejecutando la siguiente línea de código sin esperar a que se complete el tiempo de espera.

Console.log('Primer tiempo muerto completado') se ejecutará después de 2 segundos.

Mientras tanto el guion continúa ejecutando la siguiente línea de código sin causar ningún bloqueo o 'congelamiento'.

1. Programación síncrona y asíncrona

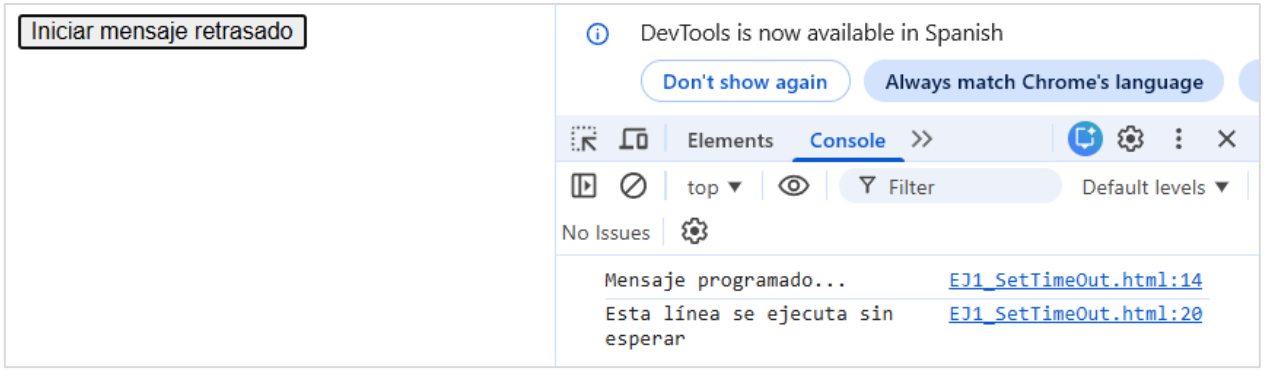
EJERCICIO 1: PROGRAMACIÓN ASÍNCRONA CON SETTIMEOUT

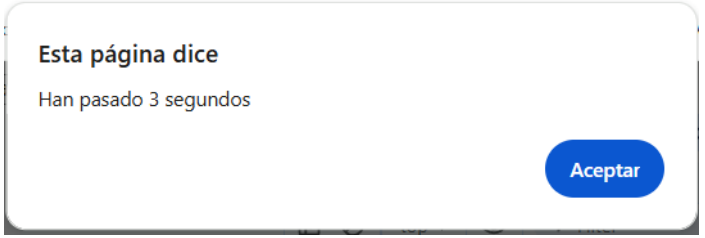
Crea una página con un botón que diga “Iniciar mensaje retrasado”.

Al hacer clic, debe mostrar en consola “Mensaje programado…”.

Programar con setTimeout que, pasados 3 segundos, se muestre un alert con el texto “Han pasado 3 segundos”.

Demuestra que es asíncrono escribiendo también inmediatamente en consola “Esta línea se ejecuta sin esperar”.





EJERCICIO 1: PROGRAMACIÓN ASÍNCRONA CON SETTIMEOUT - SOLUCIÓN

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Ejercicio setTimeout sencillo</title>
</head>
<body>
  <button id="btn">Iniciar mensaje retrasado</button>

  <script>
    const boton = document.getElementById('btn');
    boton.addEventListener('click', function () {
      console.log('Mensaje programado...');
      setTimeout(function () {
        alert('Han pasado 3 segundos');
      }, 3000);
      console.log('Esta línea se ejecuta sin esperar');
    });
  </script>
</body>
</html>
```

2. Callbacks

En JS la programación asíncrona se puede utilizar a través de distintas técnicas.

Uno de los métodos más comunes es el uso de las **callbacks** o **retrollamadas**.

FUNCIÓN CALLBACK – función que se pasa como argumento a otra función y se ejecuta después de que se haya completado la ejecución de la primera función.

En JS, las callbacks, se suelen utilizar para obtener datos de un servidor, esperar el ingreso de datos del usuario o manejar eventos.

Callbacks
https://www.w3schools.com/js/js_callback.asp

2. Callbacks

Ejemplo: FUNCIÓN CALLBACK

```
// Declarar función
function obtenerDatos(callback) {
  setTimeout(() => {
    const datos = {nombre: "John", edad: 30};
    callback(datos);
  }, 3000);
}

// Ejecutar función con una callback
obtenerDatos(function(datos) {
  console.log(datos);
});

console.log("Se están obteniendo los datos...");
```

```
Se están obteniendo los datos...
{nombre: "John", edad: 30}
```

Tenemos una función flecha que usa el método **setTimeout** para simular una operación asíncrona. La función recibe una **callback** como argumento.

La función **callback** recibe los datos recuperados por la función después de que se haya completado el tiempo de espera.

El método **setTimeout** se usa para ejecutar la **callback** después de un tiempo especificado (3 segundos). La **callback** se ejecutará de forma asíncrona, por tanto, el programa ejecutará la siguiente línea de código sin espera a que se complete el tiempo de espera.

Por tanto, se ejecuta primero la línea de obteniendo los datos mientras pasan los los 3 segundos del **setTimeout**.

2. Callbacks

EJERCICIO 2: PROGRAMACIÓN ASÍNCRONA CON CALLBACKS

Crea una función tareaAsincrona que reciba un mensaje y una función callback.

Espere 2 segundos usando setTimeout.

Después de esos 2 segundos, muestre en consola el mensaje y llame al callback.

El callback debe mostrar en consola: "Tarea terminada".

EJERCICIO 2: PROGRAMACIÓN ASÍNCRONA CON CALLBACKS - SOLUCIÓN

```
1  <!DOCTYPE html>
2  <html lang="es">
3  <head>
4    <meta charset="UTF-8">
5    <title>Callback asíncrono sencillo</title>
6  </head>
7  <body>
8    <script>
9      // Función asíncrona con callback
10     function tareaAsincrona(mensaje, callback) {
11       setTimeout(function () {
12         console.log(mensaje);          // Se ejecuta después de 2 segundos
13         callback();                    // Llamamos al callback
14       }, 2000);
15     }
16     console.log('Inicio');
17     tareaAsincrona('Procesando datos...', function () {
18       console.log('Tarea terminada');
19     });
20     console.log('Fin inmediato');
21   </script>
22 </body>
23 </html>
```

```
Inicio                          UD9_EJ2_CallBack.html:17
Fin inmediato                   UD9_EJ2_CallBack.html:23
Procesando datos...             UD9_EJ2_CallBack.html:12
Tarea terminada                 UD9_EJ2_CallBack.html:20
```

3. Callbacks Hell

Los **callbacks** permiten manejar operaciones asíncronas. Cuando se anidan muchos callbacks, el código se vuelve complejo y difícil de leer y de entender.

CALLBACK HELL – encadenación de múltiples callbacks uno tras otro, creando una estructura piramidal de indentación.

Ejemplo: CALLBACK HELL

```
obtenerDatos(function(a) {
  obtenerMasDatos(a, function(b) {
    obtenerAunMasDatos(b, function(c) {
      obtenerAunMasYMasDatos(c, function(d) {
        obtenerDatosFinales(d, function(datosFinales) {
          console.log(datosFinales);
        });
      });
    });
  });
});
```

La función **obtenerDatos** recibe un **callback** como argumento y se ejecuta después de recopilar los datos.

El **callback** toma los datos y llama a la función **obtenerMasDatos** que también recibe un **callback** como argumento y así sucesivamente.

Este anidamiento puede hacer que el código seda difícil de mantener y de identificar la estructura general del código.

Callbacks hell
https://dev.to/bryanherreradev/como-arreglar-el-callback-hell-en-javascript-4bdi

3. Callbacks Hell

EJERCICIO 3: PROGRAMACIÓN ASÍNCRONA CON CALLBACKS ENCADENADOS

Ejercicio encadenando dos tareas asíncronas con callbacks: simula cargar datos y procesarlos secuencialmente.

Usando la función tareaAsincrona del ejercicio anterior, encadena dos tareas:

- Tarea 1: Espera 2 segundos, muestra "Datos cargados", y llama a un callback para continuar.
- Tarea 2 (dentro del callback de la Tarea 1): Espera 1.5 segundos, muestra "Datos procesados", y llama a un callback final.

```
Inicio                  UD9_EJ3_CallBack_Hell.html:16
Fin inmediato           UD9_EJ3_CallBack_Hell.html:25
Datos cargados          UD9_EJ3_CallBack_Hell.html:11
Datos procesados        UD9_EJ3_CallBack_Hell.html:11
¡Proceso completo!      UD9_EJ3_CallBack_Hell.html:21
```

3. Callbacks Hell

EJERCICIO 3: PROGRAMACIÓN ASÍNCRONA CON CALLBACKS ENCADENADOS - SOLUCIÓN

```
1   <!DOCTYPE html>
2   <html lang="es">
3   <head>
4     <meta charset="UTF-8">
5     <title>Encadenar dos callbacks</title>
6   </head>
7   <body>
8     <script>
9       function tareaAsincrona(mensaje, callback) {
10        setTimeout(function () {
11          console.log(mensaje);
12          callback();  // Continúa la cadena
13        }, arguments[2] || 2000);  // Tiempo opcional (por defecto 2s)
14      }
15      console.log('Inicio');
16      tareaAsincrona('Datos cargados', function () {
17        // Tarea 2 dentro del callback de la Tarea 1
18        tareaAsincrona('Datos procesados', function () {
19          console.log('¡Proceso completo!');
20        }, 1500);  // 1.5 segundos para esta tarea
21      });
22      console.log('Fin inmediato');
23    </script>
24  </body>
25  </html>
```

```
Inicio                  UD9_EJ3_CallBack_Hell.html:16
Fin inmediato           UD9_EJ3_CallBack_Hell.html:25
Datos cargados          UD9_EJ3_CallBack_Hell.html:11
Datos procesados        UD9_EJ3_CallBack_Hell.html:11
¡Proceso completo!      UD9_EJ3_CallBack_Hell.html:21
```

Cada callback anida el siguiente para forzar el orden secuencial.

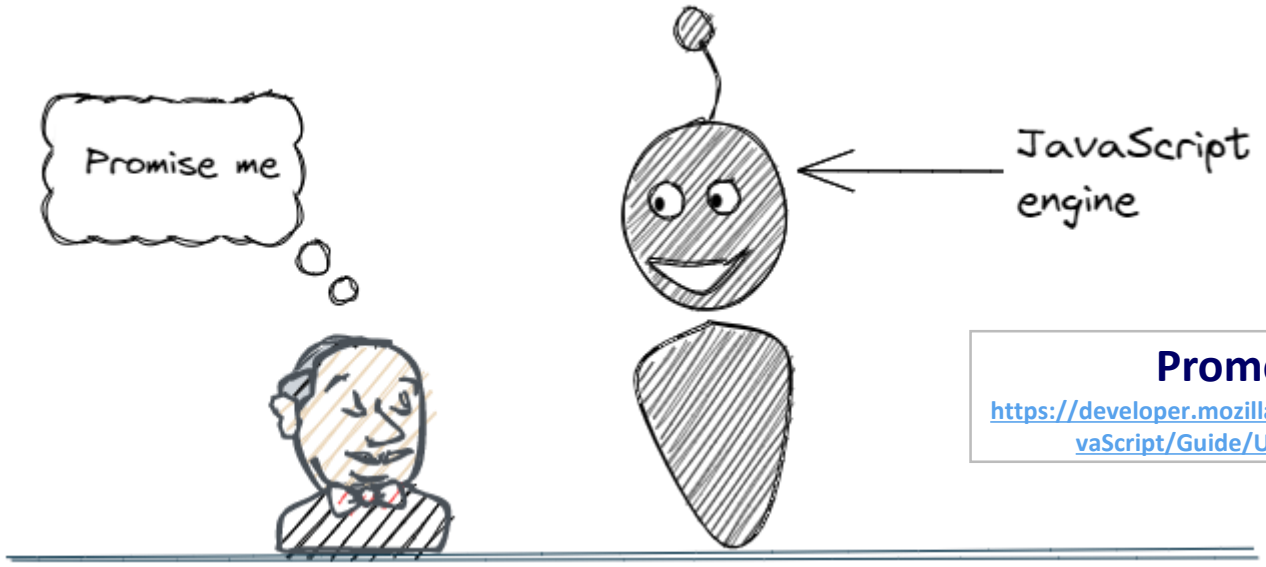
Al añadir más tareas el proceso se va complicando.

4. Promesas

Para evitar los **callback hell** se pueden utilizar las **promesas**.

PROMESA – en JS es un marcador de posición para un valor o acción futura. Es una forma de manejar las operaciones asíncronas de una manera más organizada. Tiene la misma utilidad que un **callback**, pero además de ofrecer capacidades adicionales tiene una sintaxis más legible.

Al crear una promesa, se le está diciendo al motor de JS que 'prometa' realizar una acción específica y notificarte una vez que se haya completado o fallado.



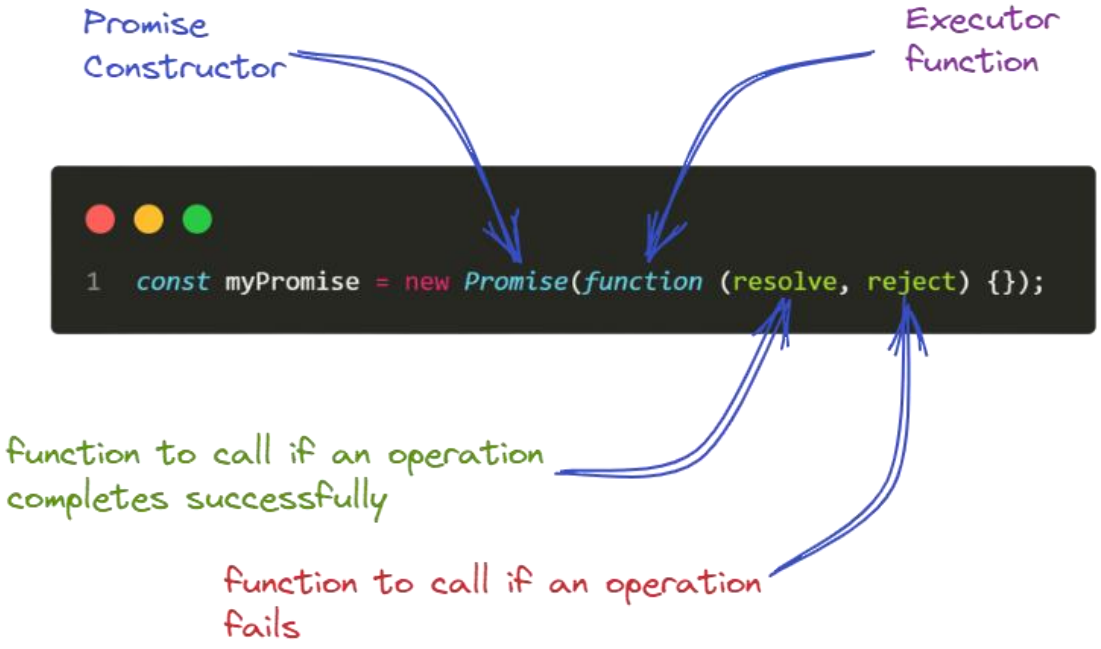
Promesas
https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Using_promises

4. Promesas

Las funciones callback se adjuntan a las promesas para manejar el resultado de la acción. Estas callbacks serán llamadas cuando la promesa se cumpla (acción completada exitosamente) o cuando se rechace (acción fallida).

Para crear una promesa, deberás crear una nueva instancia del **objeto Promise** llamando al **constructor Promise**.

El constructor toma un solo argumento: una función llamada **executor**. La función "executor" es llamada inmediatamente cuando se crea la promesa, y toma dos argumentos: una función de resolución (**función resolve**) y una función de rechazo (**función reject**).



DECLARAR UNA PROMESA

```
// Inicializar una promesa
const myPromise = new Promise(function(resolve, reject) => {})
```

```
console.log(myPromise);
```

```
▼ Promise {<pending>} i                                    script.js:10
  ▶ [[Prototype]]: Promise
    [[PromiseState]]: "pending"
    [[PromiseResult]]: undefined
```

La promesa tiene un estado pendiente y un valor indefinido. Esto se debe a que aún no se ha realizado ninguna configuración para el objeto de la promesa, por lo que va a permanecer indefinidamente en un estado pendiente sin ningún valor o resultado.

Ejemplo: PROMESA

```
const myPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve("¡Saludos desde la promesa!");
    }, 2000);
});
```

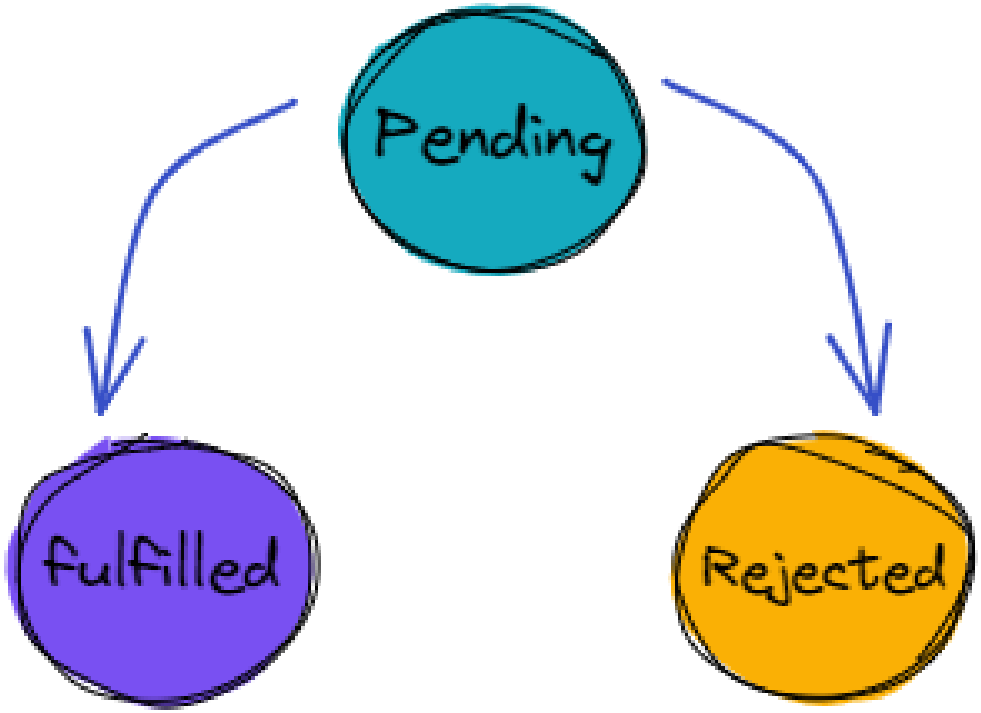
Al inspeccionar el objeto myPromise, vemos que su estado es cumplida (fulfilled) y un valor que corresponde a la cadena de caracteres que se pasó en resolve.

4. Promesas

Una promesa tiene tres estados posibles:

1) **Pendiente (pending)** – es el estado inicial, ni cumplida ni rechazada.
2) **Cumplida (fulfilled)** – la operación se completó con éxito.
3) **Rechazada (rejected)** – la operación falló.

Se dice que una promesa está resuelta tanto si se completa con éxito como si se rechaza.



4. Promesas

UTILIZAR LAS PROMESAS

Para utilizar una promesa los pasos son los siguientes:

1) **Obtener una referencia a una promesa:** Para utilizar una promesa, primero necesitas obtener una referencia a ella. Por ejemplo: el objeto myPromise.

2) **Adjuntar callbacks a una promesa:** Una vez que tienes una referencia, puedes adjuntar funciones de callback utilizando los métodos **.then** y **.catch**. El método **.then** se llama cuando una promesa se cumple y el método **.catch** se llama cuando una promesa se rechaza.

3) **Esperar a que la promesa se resuelva:** Una vez que has adjuntado los callbacks a la promesa, puedes esperar a que la promesa se cumpla o se rechace.

```
myPromise
    .then((resultado) => {
        console.log(resultado);
    })
    .catch((error) => {
        console.log(error);
    });
```

Una vez que se cumple la promesa, se llamará al método de callback **.then** con el valor resuelto.

Si la promesa se rechaza, se llamará al método **.catch** con un mensaje de error.

También puedes agregar el método **.finally()** , que se llamará después de que una promesa se resuelva. Esto significa que .finally() se invocará independientemente del estado de una promesa (ya sea resuelta o rechazada).

EJERCICIO 4: PROGRAMACIÓN ASÍNCRONA CON PROMESAS

Crea una función simularConsultaServidor() que:

- Devuelva una Promise.
- Espere 2 segundos usando setTimeout.
- Use Math.random():
  a) Si el número es mayor o igual que 0.5, llame a resolve("Datos recibidos").
  b) Si es menor que 0.5, llame a reject("Error en el servidor").

4. Promesas

EJERCICIO 4: PROGRAMACIÓN ASÍNCRONA CON PROMESAS - SOLUCIÓN

```html
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Ejercicio sencillo con Promesas</title>
</head>
<body>
  <script>
    // Función que devuelve una Promesa
    function simularConsultaServidor() {
      return new Promise((resolve, reject) => {
        setTimeout(() => {
          const numero = Math.random(); // entre 0 y 1
          console.log('Número aleatorio:', numero);
          if (numero >= 0.5) {
            resolve('Datos recibidos correctamente');
          } else {
            reject('Error en el servidor');
          }
        }, 2000); // 2 segundos
      });
    }
    console.log('Iniciando consulta...');
    simularConsultaServidor()
      .then((mensajeOk) => {
        console.log('Éxito:', mensajeOk);
      })
      .catch((mensajeError) => {
        console.error('Fallo:', mensajeError);
      });
    console.log('Petición enviada');
  </script>
</body>
</html>
```

```
Iniciando consulta...                          UD9_EJ4_Promesas.html:23
Petición enviada                               UD9_EJ4_Promesas.html:31
Número aleatorio:                              UD9_EJ4_Promesas.html:14
0.9538062641705067
Éxito: Datos recibidos                         UD9_EJ4_Promesas.html:26
correctamente
```

```
Iniciando consulta...                          UD9_EJ4_Promesas.html:23
Petición enviada                               UD9_EJ4_Promesas.html:31
Número aleatorio:                              UD9_EJ4_Promesas.html:14
0.32744061925256973
▶ Fallo: Error en el                           UD9_EJ4_Promesas.html:29
servidor
```

4. Promesas

ENCADENAR PROMESAS

El encadenamiento de promesas es un patrón que permite manejar operaciones asíncronas de forma clara y fácil de entender.

El patrón implica conectar múltiples promesas en una secuencia, donde el resultado de una promesa se pasa como argumento a la siguiente promesa.

La vinculación de las promesas se realiza mediante el uso del método **then()**. Este método utiliza una función de callback como argumento y devuelve una nueva promesa. La nueva promesa se resuelve con el valor devuelto por la función de callback.

```
fetch('https://example.com/data')
    .then(response => response.json())
    .then(data => processData(data))
    .then(processedData => {
        //haz algo con la información procesada
    })
    .catch(error => console.log(error))
```

Los métodos .then se ejecutan de manera síncrona y en orden, cada uno esperando que el anterior se resuelva. El valor de cada then se pasa como argumento al siguiente.

La **primera promesa**, es a función fetch, que está recuperando datos de un servidor.

La **segunda promesa** está analizando la respuesta como json.

La **tercera promesa** está procesando la información

La **cuarta promesa** está realizando una acción con la información obtenida.

El método **.catch** al final de la cadena manejará cualquier error que ocurra en cualquiera de las anteriores promesas.

EJERCICIO 5: ENCADENAR PROMESAS

Crea una función esperarYProcesar(valor, factor, tiempo) que devuelva una Promise.

Se inicia con new Promise que devuelve 5 tras 1 segundo y que encadena con .then():

- Primer .then: Espera 1.5s, multiplica por 2, muestra resultado.
- Segundo .then: Espera 1s, multiplica por 3, muestra resultado final.

Muestra "Inicio cadena" antes y "Cadena enviada" después para ver la asincronía.

```
Inicio cadena      UD9_EJ5_Encadenar_Promesas.html:19
Cadena enviada     UD9_EJ5_Encadenar_Promesas.html:38
Resultado: 5       UD9_EJ5_Encadenar_Promesas.html:24
Resultado: 10      UD9_EJ5_Encadenar_Promesas.html:13
Resultado: 30      UD9_EJ5_Encadenar_Promesas.html:13
```

EJERCICIO 5: ENCADENAR PROMESAS - SOLUCIÓN

```
Inicio cadena       UD9_EJ5_Encadenar_Promesas.html:19
Cadena enviada      UD9_EJ5_Encadenar_Promesas.html:38
Resultado: 5        UD9_EJ5_Encadenar_Promesas.html:24
Resultado: 10       UD9_EJ5_Encadenar_Promesas.html:13
Resultado: 30       UD9_EJ5_Encadenar_Promesas.html:13
```

```html
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Encadenar Promesas sencillo</title>
</head>
<body>
  <script>
    function esperarYProcesar(valor, factor, tiempo) {
      return new Promise((resolve) => {
        setTimeout(() => {
          const resultado = valor * factor;
          console.log(`Resultado: ${resultado}`);
          resolve(resultado);
        }, tiempo);
      });
    }
    console.log('Inicio cadena');
    new Promise((resolve) => {
      setTimeout(() => {
        resolve(5);
        console.log(`Resultado: 5`);
        }, 1000);
    })
    .then((valor) => {
      return esperarYProcesar(valor, 2, 1500);  // Espera 1.5s, multiplica x2
    })
    .then((valor) => {
      return esperarYProcesar(valor, 3, 1000);  // Espera 1s, multiplica x3
    })
    .catch((error) => {
      console.error('Error en la cadena:', error);
    });
    console.log('Cadena enviada');
  </script>
</body>
</html>
```

PROMISE.ALL

El método **Promise.all()** recibe un arreglo de promesas como argumento y retorna una única promesa que se cumple solamente cuando todas las promesas del arreglo se hayan cumplido. Se suele utilizar cuando se espera la resolución de muchas promesas antes de realizar una acción.

```
let promesa1 = fetch('https://jsonplaceholder.typicode.com/posts/1');
let promesa2 = fetch('https://jsonplaceholder.typicode.com/posts/2');
let promesa3 = fetch('https://jsonplaceholder.typicode.com/posts/3');
```

Promesa1, promesas2 y promesa3 son promesas que obtiene información de 3 urls distintas.

Se puede utilizar Promise.all([promesa1, promesas2, promesa3]) para espera a que se resuelvan todas las promesas antes de realizar una acción con los datos:

```
Promise.all([promise1, promise2, promise3])
.then((values) => {
  console.log(values);
})
```

```
▼ (3) [Response, Response, Response] i
  ▶ 0: Response {type: 'cors', url: 'https://jsonplaceholder.typicode.com/posts/1',
  ▶ 1: Response {type: 'cors', url: 'https://jsonplaceholder.typicode.com/posts/2',
  ▶ 2: Response {type: 'cors', url: 'https://jsonplaceholder.typicode.com/posts/3',
    length: 3
  ▶ [[Prototype]]: Array(0)
```

Promise.all() recibe un arreglo de promesas como argumento y devuelve una nueva promesa.

El método **then** se llama en la promesa devuelta para imprimir en consola los resultados de todas las promesas que se pasaron como argumento, en el orden en que se pasaron a **Promise.all().**

EJERCICIO 6: PROMISE.ALL

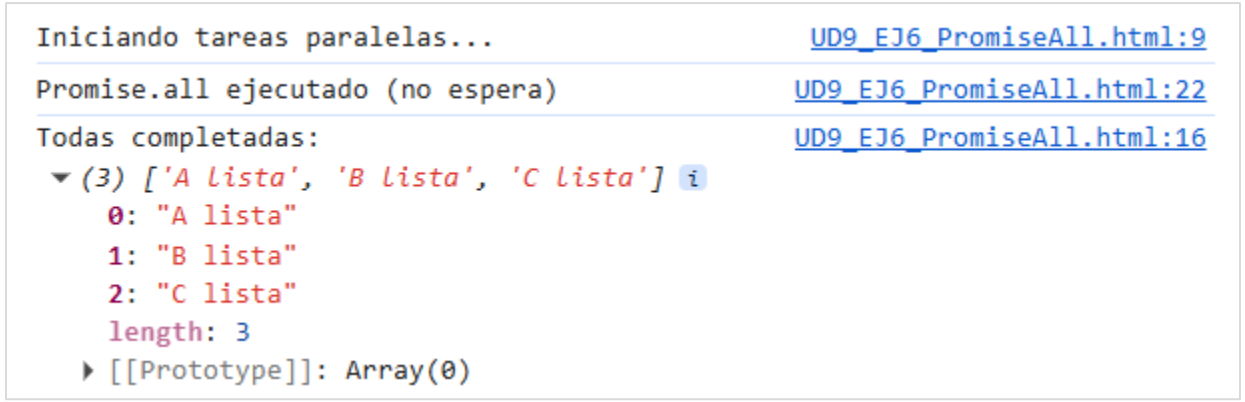
Ejecutar tres tareas en paralelo y esperar a que todas terminen:

- Tarea A: Espera 1 segundo, resuelve con "A lista".
- Tarea B: Espera 2 segundos, resuelve con "B lista".
- Tarea C: Espera 1.5 segundos, resuelve con "C lista".

Mostrar un array con los tres resultados en el orden original.

Agrega .catch() para errores.

NOTA: Promise.all() se resuelve cuando todas terminan (~2 segundos, la más lenta) o falla si cualquiera falla.



4. Promesas

```html
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Promise.all sencillo</title>
</head>
<body>
  <script>
    console.log('Iniciando tareas paralelas...');
    // Tres promesas independientes
    const tareaA = new Promise(resolve => setTimeout(() => resolve('A lista'), 1000));
    const tareaB = new Promise(resolve => setTimeout(() => resolve('B lista'), 2000));
    const tareaC = new Promise(resolve => setTimeout(() => resolve('C lista'), 1500));
    Promise.all([tareaA, tareaB, tareaC])
      .then(resultados => {
        console.log('Todas completadas:', resultados);
        // ['A lista', 'B lista', 'C lista'] - orden mantenido
      })
      .catch(error => {
        console.error('Alguna tarea falló:', error);
      });
    console.log('Promise.all ejecutado (no espera)');
  </script>
</body>
</html>
```

```
Iniciando tareas paralelas...
Promise.all ejecutado (no espera)
Todas completadas:
▼ (3) ['A lista', 'B lista', 'C lista'] i
    0: "A lista"
    1: "B lista"
    2: "C lista"
    length: 3
  ▶ [[Prototype]]: Array(0)
```

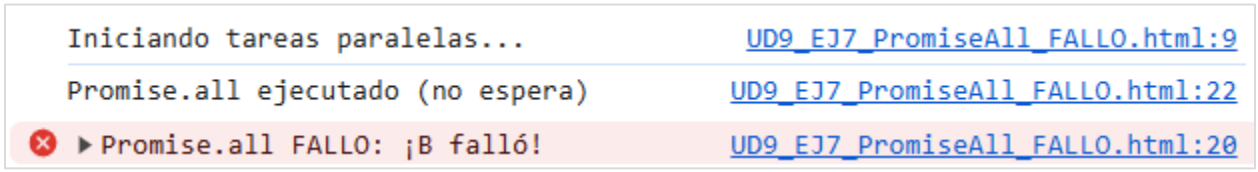
Promise.all() es perfecto para cargar múltiples recursos (imágenes, datos API) en paralelo y procesarlos cuando todos estén listos.

EJERCICIO 6: PROMISE.ALL - SOLUCIÓN

EJERCICIO 7: PROMISE.ALL CON FALLO

Modifica el ejercicio anterior para que la Tarea B falle:

NOTA: Si una promesa en Promise.all() falla, toda la operación se rechaza inmediatamente con el error de esa promesa, ignorando las demás.



4. Promesas

```
Iniciando tareas paralelas...
Promise.all ejecutado (no espera)
▶ Promise.all FALLO: ¡B falló!
```

```html
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Promise.all sencillo</title>
</head>
<body>
  <script>
    console.log('Iniciando tareas paralelas...');
    // Tres promesas independientes
    const tareaA = new Promise(resolve => setTimeout(() => resolve('A lista'), 1000));
    const tareaB = new Promise((resolve, reject) => setTimeout(() => reject('¡B falló!'), 2000));
    const tareaC = new Promise(resolve => setTimeout(() => resolve('C lista'), 1500));
    Promise.all([tareaA, tareaB, tareaC])
      .then(resultados => {
        console.log('Todas completadas:', resultados);
        // ['A lista', 'B lista', 'C lista'] - orden mantenido
      })
      .catch(error => {
        console.error('Promise.all FALLO:', error);     // Se ejecuta inmediatamente
    });
    console.log('Promise.all ejecutado (no espera)');
  </script>
</body>
</html>
```

EJERCICIO 7: PROMISE.ALL CON FALLO - SOLUCIÓN

5. Funciones asíncronas

Para escribir código asíncrono también podemos utilizar la función Async/Await:

- **Async** – palabra clave que se utiliza para declarar una función asíncrona.
- **Await** – palabra clave que se utiliza dentro de la función

```
async function obtenerDatos() {
  const respuesta = await fetch('https://jsonplaceholder.typicode.com/posts/1');
  const datos = await respuesta.json();
  console.log(datos);
}

obtenerDatos();
```

Se declara la función obtenerDatos como una función asincróna utilizando la palabra clave async.

Dentro de la función asíncrona, utilizamos la palabra clave await para esperar a que se complete la función fetch y recuperar algunos datos de una API.

Una vez que se obtienen los datos, volvemos a usar await para esperar y analizar los datos recuperados como JSON.

Por último, imprimimos los datos en la consola.

Async/await
https://es.javascript.info/async-await

EJERCICIO 8: ASYNC / AWAIT

Simular una secuencia de tareas asíncronas que se leen como código síncrono.

Crea una función async llamada procesoCompleto() que:

- Muestre "Paso 1: Iniciando...".
- Espere 2 segundos simulando una consulta.
- Muestre "Paso 2: Datos recibidos".
- Espere 1 segundo simulando procesamiento.
- Muestre "Paso 3: ¡Completado!".
- Llama a procesoCompleto() desde una función async principal y usa try/catch para errores.

```
Iniciando proceso...                    UD9_EJ8_Async_Await.html:28
Paso 1: Iniciando...                    UD9_EJ8_Async_Await.html:15
main() llamada (no espera)              UD9_EJ8_Async_Await.html:33
Paso 2: Datos recibidos                 UD9_EJ8_Async_Await.html:18
Paso 3: ¡Completado!                    UD9_EJ8_Async_Await.html:21
Proceso terminado                       UD9_EJ8_Async_Await.html:30
```

5. Funciones asíncronas

```html
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Async/Await sencillo</title>
</head>
<body>
  <script>
    // Función auxiliar para esperas
    function esperar(tiempo) {
      return new Promise(resolve => setTimeout(resolve, tiempo));
    }
    async function procesoCompleto() {
      try {
        console.log('Paso 1: Iniciando...');
        await esperar(2000);  // Espera 2 segundos

        console.log('Paso 2: Datos recibidos');
        await esperar(1000);  // Espera 1 segundo

        console.log('Paso 3: ¡Completado!');
      } catch (error) {
        console.error('Error:', error);
      }
    }
    // Función principal async
    async function main() {
      console.log('Iniciando proceso...');
      await procesoCompleto();
      console.log('Proceso terminado');
    }
    main();
    console.log('main() llamada (no espera)');
  </script>
</body>
</html>
```

```
Iniciando proceso...
Paso 1: Iniciando...
main() llamada (no espera)
Paso 2: Datos recibidos
Paso 3: ¡Completado!
Proceso terminado
```

await pausa la función **async** hasta que la promesa se resuelve, pero no bloquea el resto del navegador.

Es sintácticamente mucho más legible que callbacks o .then() encadenados.

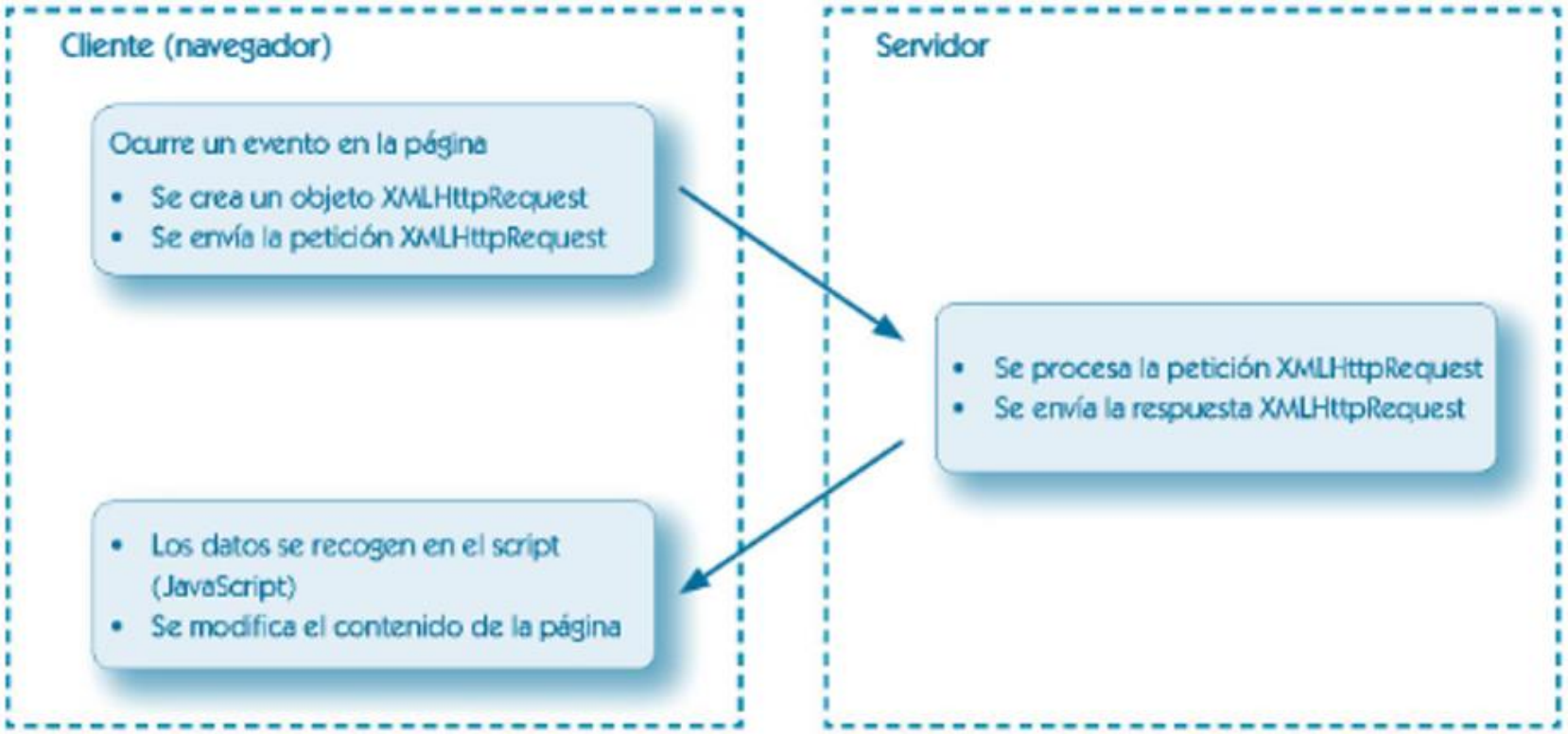
EJERCICIO 8: ASYNC / AWAIT - SOLUCIÓN

6. Ajax

AJAX (Asynchronous JavaScript and XML) – técnica web que permite actualizar contenido dinámicamente sin recargar la página completa.

Aunque en el nombre se incluye XML, este ya no se utiliza como lenguaje de intercambio de datos, sino que se utiliza JSON.

Esquema de funcionamiento de una web que utilice AJAX:



6. Ajax

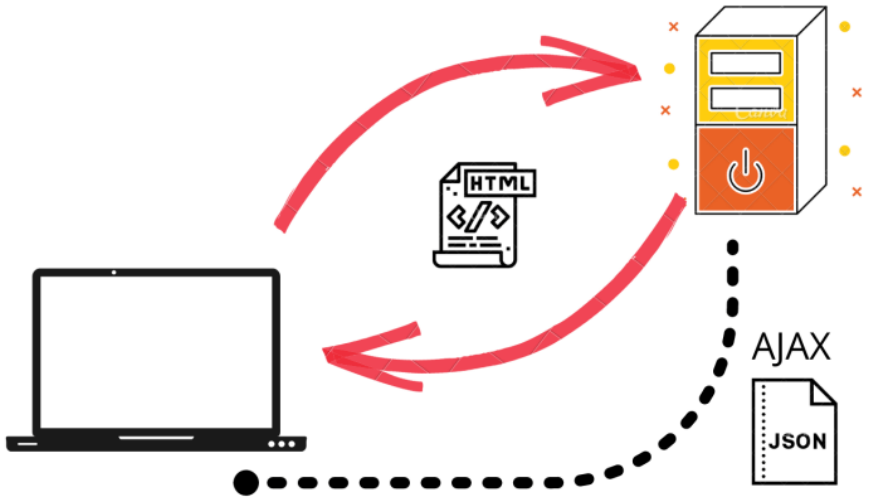
Las ventajas y desventajas de AJAX son:

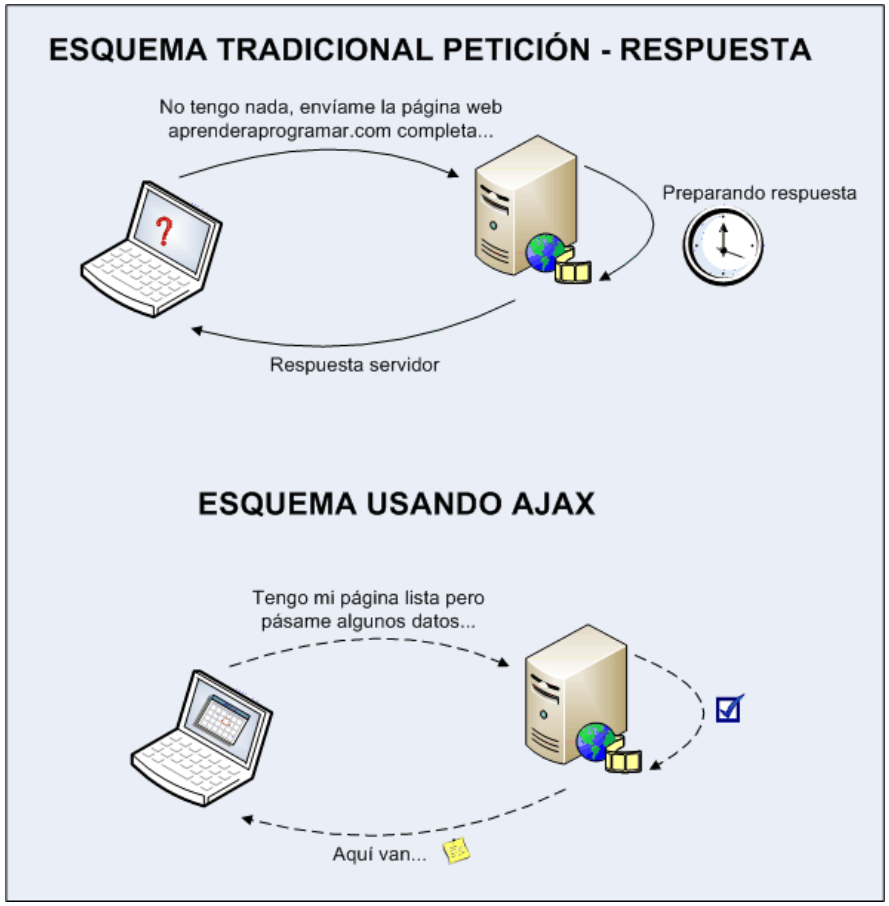
Ventajas	Desventajas
■ Simplicidad para gestionar la comunicación por medio de API de terceros. ■ Eficiencia en el intercambio de datos. ■ Favorece la creación de más servicios independientes de la plataforma. ■ Mejora la compatibilidad de servicios al ser una tecnología que no depende del lenguaje de programación del lado servidor. ■ Mejor rendimiento de las aplicaciones al ejecutar su lógica en segundo plano. ■ Gestión inteligente de formularios y validación más robusta. ■ Resolución de problemas complejos de forma sencilla.	■ Cierta complejidad a nivel de programación, solo recomendada para programadores con bastante experiencia. ■ El intercambio de datos no es más seguro que en escenarios tradicionales. ■ Peor posicionamiento en buscadores de los contenidos que incorporan de forma asíncrona. ■ Incrementa la carga de trabajo de los servidores. ■ Se pierden referencias en el historial de navegación.

¿CÓMO FUNCIONA AJAX?

El funcionamiento básico de AJAX implica:

1) Detectar un evento (clic, formulario, etc.)
2) Crear una petición asíncrona al servidor
3) Procesar la respuesta (JSON, XML o texto)
4) Actualizar el DOM dinámicamente





TRABAJAR CON AJAX EN VISUAL STUDIO CODE

1) Instalar la extensión **Live Server** – crea un servidor local con recarga automática
2) Crear el proyecto:
   I. Crear la carpeta AJAX
   II. Crear archivo index.html (estructura básica con ! + Tab)
   III. Crear archivo datos.txt (contenido: Datos cargado con AJAX!)
   IV. Crear archivo script.js (enlazar en HTML: <script src="script.js"></script>)

EJEMPLO CÓDIGO CON XMLHttpRequest():

Código HTML, enlazando con el js:

```
1   <!DOCTYPE html>
2   <html lang="es">
3   <head>
4       <meta charset="UTF-8">
5       <title>Proyecto AJAX en VS Code</title>
6   </head>
7   <body>
8       <h1>Ejemplo AJAX</h1>
9       <button onclick="cargarDatos()">Cargar Datos</button>
10      <div id="resultado">Cargando...</div>
11      <script src="script.js"></script>
12  </body>
13  </html>
```

EJEMPLO CÓDIGO CON XMLHttpRequest():

Código JS:

index.html | script.js | Open with Live Server

```
1   function cargarDatos() {
2       const xhr = new XMLHttpRequest();
3       xhr.onreadystatechange = function() {
4           if (this.readyState === 4) {
5               if (this.status === 200) {
6                   document.getElementById('resultado').innerHTML = this.responseText;
7               } else {
8                   document.getElementById('resultado').innerHTML = 'Error: ' + this.status;
9               }
10          }
11      };
12      xhr.open('GET', 'datos.txt', true);
13      xhr.send();
14  }
```

Estados **readyState**:

- **0**: new
- **1**: opened (ejecutado)
- **2**: headers_received
- **3**: loading
- **4**: done (petición terminada: éxito o error)

HTTP Status Codes del servidor:

- **200 OK**: Éxito, datos válidos
- **404 Not Found**: URL no existe
- **500 Internal Server Error**: Error servidor
- **403 Forbidden**: Sin permisos

6. Ajax

Para poder empezar a trabajar con AJAX y realizar peticiones HTTP debemos utilizar las APIs.

API – servicio externo de terceros que ofrecen una interfaz que incluye una serie de operaciones y una forma concreta de comunicarse con ellos. Como resultado, las APIs ofrecen el resultado de las peticiones en formato JSON.

Inicialmente con AJAX se utilizaba el objeto **XMLHttpRequest**, que posteriormente se sustituyó por **Fetch**, que es mucho más sencillo, eficiente y manejable.

FETCH() - realiza solicitudes HTTP asíncronas (GET, POST, etc.) devolviendo una Promesa con objeto **Response**.

6. Ajax

La **API Fetch** utiliza un método (**fetch**) al que hay que indicarla la URL de destino de la petición.

El valor devuelto es una promesa exitosa si se reciben resultados del destino sin error:

```
fetch("https://servicios.ine.es/wstempus/js/ES/OPERACIONES_DISPONIBLES")
.then(respuesta=>{
    console.log(`Código respuesta: ${respuesta.status} = ${respuesta.statusText}`);
})
.catch(error=>{
    console.log(error);
});
```

En este ejemplo se realiza una petición HTTP a una URL que forma parte de la API del Instituto Nacional de Estadística.

Esta solicitud pide la lista de operaciones que se pueden solicitar al servicio.

En este caso **then** maneja un resultado que se llama respuesta y que almacena un objeto de tipo **Response** y que además representa el paquete HTTP que contiene la respuesta.

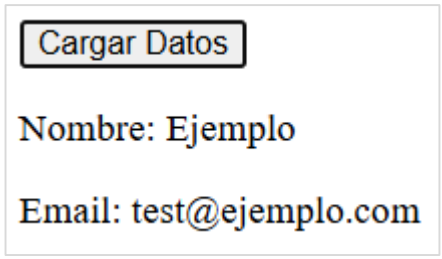
La salida permite comprobar cómo la comunicación se ha producido sin errores

```
Código respuesta: 200 = OK
>
```

HTTP Status Codes del servidor:
200 OK: Éxito, datos válidos
404 Not Found: URL no existe
500 Internal Server Error: Error servidor
403 Forbidden: Sin permisos

```html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Ejemplo Básico AJAX</title>
</head>
<body>
    <button id="cargar">Cargar Datos</button>
    <div id="resultado"></div>

    <script>
        document.getElementById('cargar').addEventListener('click', function() {
            var xhr = new XMLHttpRequest();
            xhr.open('GET', 'datos.json', true);  // true para asíncrono
            xhr.onreadystatechange = function() {
                if (xhr.readyState == 4 && xhr.status == 200) {
                    var datos = JSON.parse(xhr.responseText);
                    document.getElementById('resultado').innerHTML =
                        '<p>Nombre: ' + datos.nombre + '</p>' +
                        '<p>Email: ' + datos.email + '</p>';
                }
            };
            xhr.send();
        });
    </script>
</body>
</html>
```



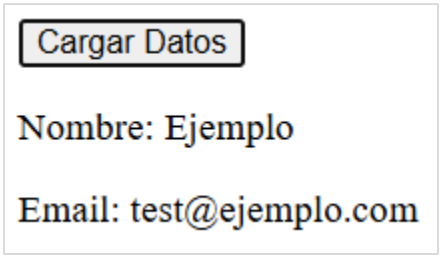
EJEMPLO: CARGA DE DATOS DE UN ARCHIVO JSON VÍA GET

XMLHttpRequest

Creamos datos.json con {"nombre": "Ejemplo", "email": "test@ejemplo.com"}

```html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Ejemplo Básico AJAX</title>
</head>
<body>
    <button id="cargar">Cargar Datos</button>
    <div id="resultado"></div>

    <script>
        document.getElementById('cargar').addEventListener('click', function() {
            fetch('datos.json')
                .then(response => response.json())
                .then(datos => {
                    document.getElementById('resultado').innerHTML =
                        '<p>Nombre: ' + datos.nombre + '</p>' +
                        '<p>Email: ' + datos.email + '</p>';
                })
        });
    </script>
</body>
</html>
```



EJEMPLO: CARGA DE DATOS DE UN ARCHIVO JSON VÍA GET

FETCH

Creamos datos.json con {"nombre": "Ejemplo", "email": "test@ejemplo.com"}

```
<script>
    document.getElementById('cargar').addEventListener('click', function() {
        var xhr = new XMLHttpRequest();
        xhr.open('GET', 'datos.json', true);  // true para asíncrono
        xhr.onreadystatechange = function() {
            if (xhr.readyState == 4 && xhr.status == 200) {
                var datos = JSON.parse(xhr.responseText);
                document.getElementById('resultado').innerHTML =
                    '<p>Nombre: ' + datos.nombre + '</p>' +
                    '<p>Email: ' + datos.email + '</p>';
            }
        };
        xhr.send();
    });
</script>
```

COMPARATIVA

XMLHttpRequest

```
<script>
    document.getElementById('cargar').addEventListener('click', function() {
        fetch('datos.json')
            .then(response => response.json())
            .then(datos => {
                document.getElementById('resultado').innerHTML =
                    '<p>Nombre: ' + datos.nombre + '</p>' +
                    '<p>Email: ' + datos.email + '</p>';
            })
    });
</script>
```

Fetch

CESUR
Tu Centro Oficial de FP

Propiedades y métodos del **objeto Response**:

Miembro	Significado
headers	Objeto que contiene las cabeceras HTTP de la respuesta.
body	Contiene los datos de la respuesta.
status	Código de la respuesta a la petición HTTP (estandarizado en el protocolo HTTP).
statusText	Texto que interpreta el código de estado.
ok	Si vale **true** quiere decir que la petición se produjo sin errores.
redirected	Si vale **true** indica que la respuesta es fruto de una redirección.
url	Dirección de la que parte la respuesta.
type	Contiene el tipo de respuesta recibida.
redirect()	Redirecciona la respuesta a otra dirección.
clone()	Clona la respuesta en otro objeto que interese.
error()	Devuelve un objeto respuesta asociado a un error de red.
text()	Toma un flujo de objeto respuesta y lo lee hasta completarlo. Devuelve una promesa que resuelta con éxito obtiene los datos de la respuesta en modo texto, siempre decodificada con UTF-8.
json()	Misma funcionalidad que **text()**, pero devuelve la respuesta en formato JSON.
blob()	Misma funcionalidad que **text()**, pero devuelve la respuesta como un objeto binario.

Códigos de estado de http

https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml

EJEMPLO RESPUESTAS EN FORMATO TEXTO:

```
fetch("https://lopegonzalez.es/servicios/vehiculos.php?matricula=1702TGG")
.then(respuesta=>respuesta.text())
.then(textoDevuelto=>{console.log(textoDevuelto)})
.catch(error=>{
    console.log(error);
});
```

En la url: https://lopegonzalez.es/servicios/vehiculos.php hay una rutina que acepta peticiones HTTP de tipo GET.

Si se pasa el parámetro matrícula de un vehículo, ofrecerá información en formato texto del vehículo en cuestión (si está registrado en la aplicación).

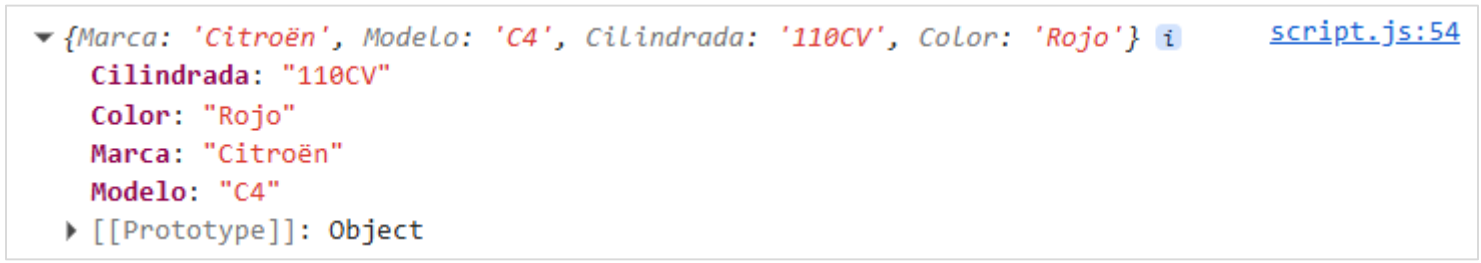
Usamos el método **test()** de la API Fetch sobre el objeto de respuesta y se obtiene texto plano como resultado de la solicitud:

```
{"Marca":"Citro\u00ebn","Modelo":"C4","Cilindrada":"110CV","Color":"Rojo"}
```

EJEMPLO RESPUESTAS EN FORMATO JSON:

```
fetch("https://lopegonzalez.es/servicios/vehiculos.php?matricula=1702TGG")
.then(respuesta=>{
    if(respuesta.ok){
        return respuesta.json();
    }
})
.then(datos=>{
    console.log(datos);
})
.catch(error=>{
    console.log(error);
});
```

En este caso, el metodo **json()** del objeto de la respuesta genera una promesa que de resolverse con éxito entrega los datos como un objeto JSON que puede manipularse fácilmente desde JavaScript:



EJERCICIO 9: FORMATEO DE RESPUESTAS

Utilizando el código del ejemplo anterior, intégralo en una pagina web que consiga representar los datos en formato HTML y aplique a posteriori estilos CSS, de manera que la marca y el modelo sean h2, el color será una caja coloreada y la cilindrada aparezca en negrita dentro de un span.

EJERCICIO 9: FORMATEO DE RESPUESTAS – SOLUCIÓN I

```html
<! DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Proyecto AJAX en VS Code</title>
    <link rel="stylesheet" href="estilos.css">
</head>
<body>
    <div id="contenedor"></div>
    <a href="#" id="boton">Mostrar datos del vehículo</a>
    <script src="script.js"></script>
</body>
</html>
```


```css
h2 { font-size: 24px; margin-bottom: 10px; }
.color {
    display: block;
    width:10px;
    padding: 10px;
    margin-bottom: 10px;
    font-weight: bold;
}
.rojo { background-color: red; color: white; }
.verde { background-color: green; color: white; }
.azul { background-color: blue; color: white; }
.cilindrada { font-weight: bold; }
span {padding: 20px;}
```

EJERCICIO 9: FORMATEO DE RESPUESTAS – SOLUCIÓN II

```
1   const boton = document.getElementById("boton");
2   const contenedor = document.getElementById("contenedor");
3
4   boton.addEventListener("click", () => {
5       fetch("https://lopegonzalez.es/servicios/vehiculos.php?matricula=1702TGG")
6       .then(respuesta=>{
7           if(respuesta.ok){
8               return respuesta.json();
9           }
10      })
11      .then(datos=>{
12          console.log(datos);
13          const marcaModelo = document.createElement("h2");
14          marcaModelo.textContent = `${datos.Marca} ${datos.Modelo}`;
15
16          const color = document.createElement("div");
17          color.classList.add("color", datos.Color);
18
19          const cilindrada = document.createElement("span");
20          color.classList.add("cilindrada");
21          cilindrada.textContent = `${datos.Cilindrada} cc`;
22
23          contenedor.appendChild(marcaModelo);
24          contenedor.appendChild(color);
25          contenedor.appendChild(cilindrada);
26      })
27      .catch(error=>{
28          console.log(error);
29      });
30  });
```

Citroën C4

110CV cc

Mostrar datos del vehículo

Cuando se realiza una petición se está indicando implícitamente el objeto **Request**, que tiene las siguientes propiedades:

Propiedad	Utilidad
url	Dirección destino a la que se realiza la petición.
method	Acepta cualquier método HTTP (**GET, POST, PUT, DELETE**…), aunque los más frecuentes son los dos primeros; si no se especifica, el valor predeterminado es **GET.**
headers	Cabeceras de la petición.
mode	Tratamiento de la directiva CORS. Existen estas opciones: ■ **no-cors:** restringe las cabeceras que se pueden usar y solo admite los métodos **GET, POST** y **PUT.** ■ **cors:** valor predeterminado que permite todas las respuestas. ■ **same-origin:** fuerza a que la respuesta sea solo del mismo dominio del que parte la petición. ■ **navigate:** la petición no es para ser usada con AJAX, sino directamente como dirección del navegador.
cache	Los navegadores normalmente cachean las respuestas para ganar en eficiencia. Pero con esta propiedad puede alterarse su funcionamiento. Las opciones permitidas son: ■ **default:** comportamiento predeterminado. Si la respuesta estaba cacheada la recupera. ■ **no-cache:** se comprueba si la respuesta es la misma, y si lo es se recoge de la caché; si no, se construye la nueva respuesta. ■ **force-cache:** se fuerza a usar la caché. Si no está presente se pide al servidor y se cachea para la siguiente ocasión. ■ **only-if-cached:** solo se usan los datos almacenados en la caché. ■ **reload:** fuerza a construir una respuesta nueva, pero aun así se cachea. ■ **no-store:** no cachea nunca la respuesta.
redirect	Permite indicar cómo se debe actuar ante las redirecciones: **follow** (se siguen), **error** (se devuelve un error), **manual** (el usuario debe intervenir).
credentials	Ofrece la posibilidad de indicar si se admiten *cookies*.
integrity	Un algoritmo criptográfico genera un *hash* para comprobar la integridad de la petición.

Todas estas propiedades del objeto **Request** pueden indicarse (nombre y valor) como segundo parámetro de fetch (el primero será la url).

```
fetch("https://lopegonzalez.es/servicios/request.php",{
        method:"POST",
        mode:"same-origin",
        cache:"reload",
        redirect:"follow",
        credentials:"include"
});
```

En las comunicaciones HTTP, tanto en las peticiones como en las respuestas incluyen en sus paquetes una serie de cabeceras.

Para indicar el **headers** se debe usar un constructor e indicar los valores de las propiedades (igual que haríamos con cualquier otro objeto):

```
let miCabecera = new Headers({
        "Content-Type":"multipart/form-data",
        "WWW-Authenticate":"Basic",
        ...
});
```

HTTP headers
https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers

En el constructor se pueden ir modificando dinámicamente los valores de **headers** usando los siguientes métodos:

Método	Utilidad
append	Añade un nuevo valor a una cabecera existente en un objeto **Headers** o añade la cabecera si no existe.
delete	Elimina una cabecera de un objeto **Headers.**
entries	Devuelve un iterador que permite recorrer los pares clave-valor contenidos en el objeto.
get	Devuelve el valor de una cabecera que se especifique.
has	Devuelve **true** si el objeto **Headers** tiene una cabecera en concreto.
keys	Devuelve un iterador que permite recorrer las claves de las cabeceras del objeto.
set	Modifica el valor de una cabecera existente en un objeto **Headers** o añade la cabecera si no existe.
values	Devuelve un iterador que permite recorrer los valores de las cabeceras del objeto.

Utilizando el método POST, tenemos menos flexibilidad al enviar datos en la petición, por eso es preferible usar el método GET.

```
https://lopegonzalez.es/servicios/vehiculos.php?matricula=1702TGG
```

En este caso se envía matrícula con el valor 1702TGG.

Si queremos enviar más datos, solo hay que encadenarlos con & incluyendo pares clave-valor:

```
vehiculos.php?matricula=1702TGG&color=azul&cilindrada=110CV
```

Normalmente se suele usar:

1. **GET** – pedir datos
2. **POST** – para enviar datos a un servidor
3. **PUT** – modificar datos
4. **DELETE** – para borrarlos

ENVIO DE DATOS DE UN FORMULARIO

Al enviar datos de un formulario sin utilizar AJAX, generalmente se usa el método POST, para que los datos no sean visibles en la URL.

Cuando se utiliza **AJAX**, no sólo debe indicarse que el método utilizado es POST, sino que debe indicar en las cabeceras de la petición que se trata de datos procedente de un formulario e incluir los datos en la propiedad **body**.

Aunque se envíen por POST, se pueden indicar internamente en la forma clave-valor:

```
fetch("https://lopegonzalez.es/servicios/nuevovehiculo.php",{
        method:"POST",
        headers:{'Content-Type':'application/x-www-form-urlencoded'},
        body:"matricula=1402LGM&marca=Peugeout&modelo=206"
});
```

6. Ajax

Existe otra forma de enviar los datos, pensado para el enviado de datos de gran tamaño, es utilizando el tipo de contenido **application/multipart/form-data** mediante el objeto **FormData**, en el que se pueden ir añadiendo los pares claves.valor del formulario:

```
let formulario = new FormData();
formulario.append("matricula","1402LGM");
formulario.append("marca","Peugeout");
formulario.append("modelo","206");
formulario.set("modelo","306");
formulario.delete("modelo");
fetch("https://lopegonzalez.es/servicios/nuevovehiculo.php",{
        method:"POST",
        headers:{'Content-Type':'application/mutipart/form-data'},
        body:formulario
});
```

Los métodos de FormData que nos permite gestionar formularios son:

1) **append()** – nos permite añadir pares clave-valor
2) **set()** – para modificar el valor de una clave
3) **delete()** – para eliminar pares clave-valor
4) **get()** – obtiene el valor de una clave
5) **has()** – comprueba si existe una clave

Objeto FormData
https://developer.mozilla.org/es/docs/Web/API/XMLHttpRequest_API/Using_FormData_Objects

El objeto se puede construir con todo el formulario de una sola vez y el constructor se encargará de gestionar los pares clave-valor internamente:

```
let datosFormulario = document.querySelector("form");
datosFormulario.addEventListener("submit",(evento)=>{
        evento.preventDefault();
        let formulario = new FormData(datosFormulario);
        fetch(datosFormulario.getAttribute("action"),{
                method:"POST",
                headers:{'Content-Type':'application/mutipart/form-data'},
                body:formulario
        });
});
```

Se utiliza un **querySelector** con la etiqueta form.

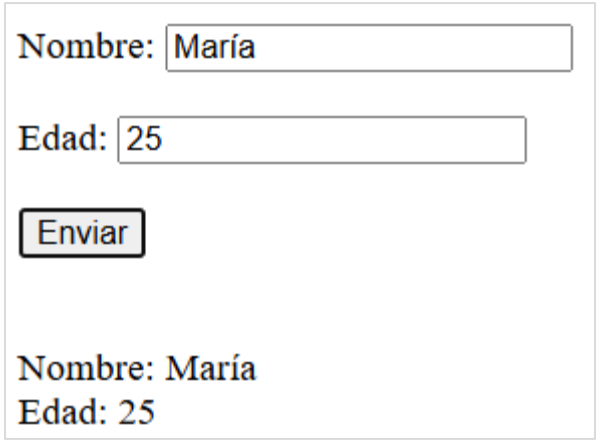
Se **captura el clic del usuario** para enviar el formulario, se **cancela su comportamiento predeterminado** y se construye la petición AJAX recogiendo del propio formulario la URL de destino.

Finalmente, con el objeto **FormData** se construyen los datos que se proporcionan al bloque body de la petición.

EJERCICIO 10: FORMDATA

Crea un formulario con dos campos Nombre y Edad y un botón de Enviar.

Crea un FormData a partir del formulario y simula el envío mostrando los datos en un div.



EJERCICIO 10: FORMDATA - SOLUCIÓN

```html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Ejemplo Básico AJAX</title>
</head>
<body>
<!-- index.html -->
<form id="miFormulario" enctype="multipart/form-data">
  <label>Nombre:</label>
  <input type="text" name="nombre" required><br><br>

  <label>Edad:</label>
  <input type="number" name="edad" required><br><br>

  <button type="submit">Enviar</button>
</form>
<div id="respuesta"></div>
<script>
  const form = document.getElementById("miFormulario");
  const divRespuesta = document.getElementById("respuesta");
  form.addEventListener("submit", async (e) => {
    e.preventDefault();
    // 1. Crear FormData a partir del formulario
    const formData = new FormData(form);
    try {
      const texto = "Nombre: " + formData.get('nombre') + "<br>Edad: " + formData.get('edad');
      divRespuesta.innerHTML = texto;
    } catch (error) {
      divRespuesta.innerHTML = "Error: " + error.message;
    }
  });
</script>
</body>
</html>
```

SOLUCIÓN CON PHP

```
try {
    // 2. Enviar con fetch
    const respuesta = await fetch("procesar.php", {
      method: "POST",
      body: formData
    });

    // 3. Leer respuesta como texto
    const texto = await respuesta.text();
    divRespuesta.innerHTML = texto;
  } catch (error) {
    divRespuesta.innerHTML = "Error: " + error.message;
  }
```

ACTIVIDAD 1: ENCUESTA INTERACTIVA CON AJAX I

Crear una encuesta interactiva utilizando AJAX.

La encuesta hará una pregunta al usuario y mostrará el resultado total de todas las preguntas realizadas hasta el momento.

La página web constará de tres partes:

1) Un fichero de texto (resultados.txt). Donde se almacenan los resultados.
2) Una página html (encuesta.html). Donde se genera la encuesta.
3) Una página php (encuesta_voto.php). Recoge el voto del usuario y calcula los porcentajes teniendo en cuenta los datos anteriores, además de publicar el resultado.

1.- El fichero de texto (resultados.txt)

Es donde se van a almacenar los resultados de la encuesta. Se va a llamar resultados.txt y se va a iniciar con la siguiente línea de texto: `0||0||0||0`

Cada uno de esos campos separados por || serán los votos que obtendrá cada una de las opciones. Conforme los usuarios vayan votando, el contenido del fichero irá cambiando: `19||3||6||0`

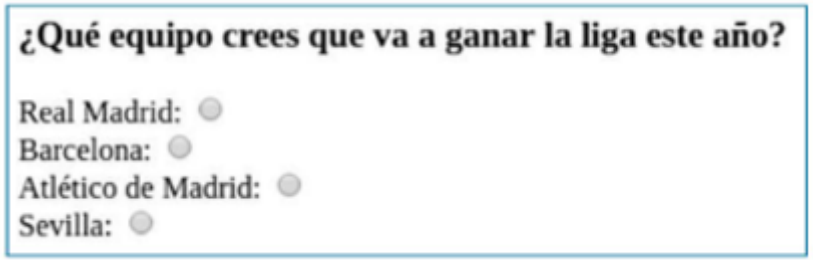
ACTIVIDAD 1: ENCUESTA INTERACTIVA CON AJAX II

2.- La página web (encuesta.html)

Consta de cuatro inputs de tipo radio que, al ser seleccionados, ejecutarán la función **getvoto().**

La función **getvoto()** realiza una llamada AJAX a la página php **encuesta_voto.php** pasándole como parámetro por GET el valor del voto emitido por el usuario.

Su aspecto será:



3.- **La página php** (**encuesta_voto.php**) – incluida

Gracias

- https://www.freecodecamp.org/espanol/news/programacion-asincrona-en-javascript-guia-para-principiantes/
- https://keepcoding.io/blog/callback-hell-en-javascript/