



Introducción



Desarrollo de Sistemas Web - Back End
Tareas Asíncronas



Tareas Asíncronas



Definición: Las tareas asíncronas en programación se refieren a operaciones que se ejecutan de manera independiente del flujo principal del programa. En lugar de bloquear el programa mientras una operación se completa, se permite que otras tareas continúen ejecutándose.

Características:

1. **No Bloqueantes:** Las operaciones asíncronas no bloquean el hilo principal, permitiendo que otras tareas se realicen en paralelo.
2. **Eficiencia:** Mejoran la eficiencia al gestionar múltiples operaciones simultáneamente, especialmente en operaciones de entrada/salida como la lectura de archivos o las solicitudes de red.
3. **Eventos y Promesas:** Utilizan mecanismos como promesas, callbacks o `async/await` para manejar la finalización de las tareas.
4. **Complejidad:** A veces pueden introducir complejidad adicional en la gestión del flujo de datos y el manejo de errores.

Ejemplos:

- Leer un archivo sin bloquear la ejecución de otras operaciones.
- Hacer una solicitud HTTP a una API mientras se realizan otras tareas en la aplicación.

La sincronía en Javascript

En JavaScript, la sincronía se refiere a la manera en que las operaciones y los eventos se ejecutan de forma secuencial, uno después del otro, en el hilo principal del programa. Sin embargo, JavaScript también permite la ejecución de operaciones asíncronas, lo que significa que ciertas tareas pueden ejecutarse en paralelo sin bloquear el flujo principal del programa. En el ejemplo donde tenemos que realizar tareas que tienen que esperar a que ocurra un determinado suceso que no depende de nosotros, y reaccionar realizando otra tarea sólo cuando dicho suceso ocurra.

Sincrónico

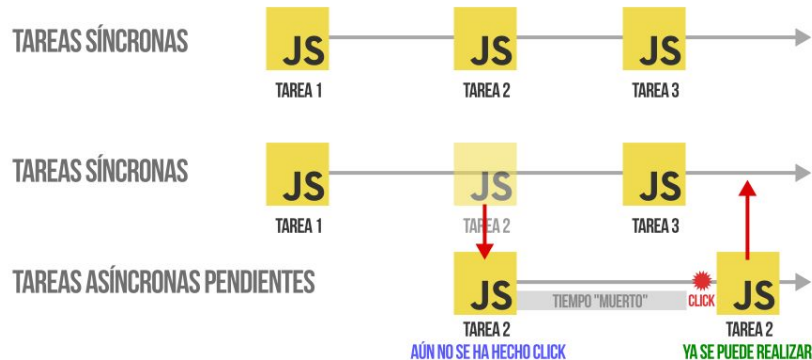
```
console.log('Inicio');  
console.log('Proceso');  
console.log('Fin');
```

Asincrónico

```
console.log('Inicio');  
  
setTimeout(() => {  
    console.log('Esto se ejecuta después de 2 segundos');  
}, 2000);  
  
console.log('Fin');
```

El término "lenguaje no bloqueante" se refiere a una forma de programación que permite realizar múltiples operaciones de manera simultánea o en paralelo sin que una tarea o proceso bloquee el progreso de otros. Este concepto es fundamental para la programación asíncrona y concurrente, especialmente en lenguajes y entornos que están diseñados para manejar operaciones de entrada/salida (I/O) y otras tareas que pueden ser lentas o impredecibles sin detener la ejecución del programa principal.

Está diseñado para trabajar de manera eficiente con operaciones asíncronas a través del Event Loop y la programación basada en callbacks, Promises y `async/await`. Aquí te explico cómo JavaScript maneja el concepto de no bloqueo y cómo esto se relaciona con la ejecución concurrente y paralela





Gestionar la asincronía, donde quizás las más populares son las siguientes:

Mediante callbacks: son funciones que se pasan como argumentos a otras funciones y se ejecutan después de que la tarea asíncrona se ha completado.

Mediante promesas: Una forma más moderna y actual de gestionar la asincronía. Las Promises representan el resultado de una operación asíncrona que puede completarse en el futuro. Permiten encadenar operaciones y manejar errores de manera más clara que los callbacks.

Mediante `async/await` proporcionan una sintaxis más legible para trabajar con Promises, permitiendo escribir código asíncrono que se asemeja al estilo de código síncrono.

Mediante top-level `await`: Una variación de la anterior, donde no es necesario usar `async` en determinados contextos.

Programación Asíncrona en JavaScript



Callbacks

Los callbacks son funciones que se pasan como argumentos a otras funciones y se ejecutan después de que la tarea asíncrona se ha completado. A pesar de ser una forma flexible y potente de controlar la asincronía, que permite realizar múltiples posibilidades, las funciones callbacks tienen ciertas desventajas evidentes.

En primer lugar, el código creado con las funciones es algo caótico y (quizás subjetivamente) algo feo. Por ejemplo, el tener que pasar un null por parámetros en algunas funciones, no es demasiado elegante.

```
console.log('Inicio');

setTimeout(() => {
  console.log('Esto se ejecuta después de 2 segundos');
}, 2000);

console.log('Fin');
```

Promesas (Promises)



Las promesas en Javascript se representan a través de un object, y cada promesa estará en un estado concreto: pendiente, aceptada o rechazada. Por norma general, las tareas asíncronas no sabemos cuánto tardarán en responder y/o procesarse, por lo que muchas veces el orden en que se resuelvan no será el mismo. Esto en algunos casos no nos importará, pero en otros sí, por lo que hay que tenerlo en cuenta. Además, cada promesa tiene los siguientes métodos, que podremos utilizar para utilizarla:

`.then(resolve)`: Ejecuta la función callback resolve cuando la promesa se cumple.

`.catch(reject)`: Ejecuta la función callback reject cuando la promesa se rechaza.

`.then(resolve, reject)`: Método equivalente a las dos anteriores en el mismo `.then()`.

`.finally(end)`: Ejecuta la función callback end tanto si se cumple como si se rechaza.

```
console.log('Inicio');

const promesa = new Promise((resolve) => {
  setTimeout(() => {
    resolve('Promesa completada');
  }, 2000);
});

promesa.then((resultado) => {
  console.log(resultado);
});

console.log('Fin');
```



Fetch()

La función `fetch()` es una API moderna en JavaScript que se utiliza para realizar solicitudes HTTP. Fue introducida en la especificación Fetch API para reemplazar a `XMLHttpRequest` y proporcionar una forma más sencilla y potente de manejar las operaciones de red. Aquí tienes una explicación detallada sobre cómo funciona `fetch()` y cómo se usa:

¿Qué es `fetch()`?

- **Definición:** `fetch()` es una función global en JavaScript que permite hacer solicitudes HTTP de manera sencilla y trabajar con respuestas asíncronas. Está basada en promesas, lo que facilita la gestión de operaciones asíncronas y permite un código más limpio y manejable.

Más adelante profundizaremos esta API



API de las promesas

El objeto `Promise` de Javascript tiene varios métodos estáticos que podemos utilizar en nuestro código. Todos devuelven una promesa y son los siguientes:

- `Promise.all(object list)`: Acepta sólo si todas las promesas del grupo se cumplen.
- `Promise.allSettled(object list)`: Acepta sólo si todas las promesas del grupo se cumplen o rechazan.
- `Promise.any(object value)`: Acepta con el valor de la primera promesa del grupo que se cumpla.
- `Promise.race(object value)`: Acepta o rechaza dependiendo de la primera promesa que se procese.
- `Promise.resolve(object value)`: Devuelve un valor envuelto en una promesa que se cumple directamente.
- `Promise.reject(object value)`: Devuelve una promesa rechazada con el valor proporcionado.



Método: Promise all ()

```
const p1 = fetch('/robots.txt');
const p2 = fetch('/index.css');
const p3 = fetch('/index.js');

Promise.all([p1, p2, p3]).then((responses) => {
  responses.forEach((response) => {
    console.log(response.status, response.url);
  });
});
```

El método `Promise.all()` funciona como un «todo o nada»: devuelve una promesa que se cumple cuando todas las promesas del array se cumplen. Si alguna de ellas se rechaza, `Promise.all()` también lo hace.



Promises.allSettled()

```
const p1 = fetch('/robots.txt');
const p2 = fetch('https://google.com/index.css');
const p3 = fetch('/index.js');

Promise.allSettled([p1, p2, p3]).then((responses) => {
  responses.forEach((response) => {
    console.log(response.status, response);
  });
});
```

El método `Promises.allSettled()` funciona como un «todas procesadas»: devuelve una promesa que se cumple cuando todas las promesas del array se hayan procesado, independientemente de que se hayan cumplido o rechazado.



Promise.any()

```
const p1 = fetch('/robots.txt');  
const p2 = fetch('/index.css');  
const p3 = fetch('/index.js');  
  
Promise.any([p1, p2, p3]).then((response) =>  
  console.log(response.status, response.url)  
);
```

El método `Promise.any()` funciona como «la primera que se cumpla»: Devuelve una promesa con el valor de la primera promesa individual del array que se cumpla. Si todas las promesas se rechazan, entonces devuelve una promesa rechazada.



Promise.race()

```
const p1 = fetch('/robots.txt');
const p2 = fetch('/index.css');
const p3 = fetch('/index.js');

Promise.race([p1, p2, p3]).then((response) =>
  console.log(response.status, response.url)
);
```

El método `Promise.race()` funciona como una «la primera que se procese»: la primera promesa del array que sea procesada, independientemente de que se haya cumplido o rechazado, determinará la devolución de la promesa del `Promise.race()`. Si se cumple, devuelve una promesa cumplida, en caso negativo, devuelve una rechazada.



Async/Await

1. **async**:

- La palabra clave **async** se usa para declarar una función asíncrona. Esto convierte a la función en una que siempre devuelve una promesa.
- Dentro de una función **async**, puedes usar **await** para esperar a que una promesa se resuelva.

2. **await**:

- La palabra clave **await** se usa dentro de una función **async** para esperar a que una promesa se resuelva. Esto hace que el código asíncrono sea más fácil de leer y escribir, ya que parece síncrono.

```
console.log('Inicio');

async function ejecutar() {
  const promesa = new Promise((resolve) => {
    setTimeout(() => {
      resolve('Promesa completada');
    }, 2000);
  });

  const resultado = await promesa;
  console.log(resultado);
}

ejecutar();

console.log('Fin');
```



Peticiones HTTP

Un navegador, durante la carga de una página, suele realizar múltiples peticiones HTTP a un servidor para solicitar los archivos que necesita renderizar en la página. Es el caso de, en primer lugar, el documento .html de la página (donde se hace referencia a múltiples archivos) y luego todos esos archivos relacionados: los ficheros de estilos .css, las imágenes .jpg, .png, .webp u otras, los scripts .js, las tipografías .ttf, .woff o .woff2, etc.

Una petición HTTP Es como suele denominarse a la acción por parte del navegador de solicitar a un servidor web un documento o archivo, etc.

Gracias a dicha petición, el navegador puede descargar ese archivo, almacenarlo en un caché temporal de archivos del navegador y, finalmente, mostrarlo en la página actual que lo ha solicitado.

Las peticiones HTTP son fundamentales para la comunicación entre clientes y servidores en aplicaciones web. A través de peticiones HTTP, un cliente puede solicitar recursos o enviar datos a un servidor, y el servidor responde con la información solicitada o el resultado de la operación.



Exploraremos cómo realizar peticiones HTTP en varios entornos:

1. **Peticiones HTTP en el Navegador con JavaScript:** Utilizaremos `fetch`, que es una API moderna para hacer solicitudes HTTP.
2. **Peticiones HTTP en el Servidor con Node.js:** Utilizaremos la biblioteca `axios` o el módulo `http` nativo.

Métodos de peticiones AJAX



AJAX (Asynchronous JavaScript and XML) permite a las páginas web hacer peticiones al servidor y actualizar partes de la página sin recargarla completamente.

Aunque AJAX tradicionalmente utilizaba el objeto XMLHttpRequest, hoy en día es común usar fetch para hacer peticiones asíncronas de manera más sencilla. AJAX

Esta modalidad se basa en que la petición HTTP se realiza desde Javascript, de forma transparente al usuario, descargando la información y pudiendo tratarla sin necesidad de mostrarla directamente en la página.

Esto produce un interesante cambio en el panorama que había entonces, puesto que podemos hacer actualizaciones de contenidos de forma parcial, de modo que se actualice una página «en vivo», sin necesidad de recargar toda la página, sino solamente actualizado una pequeña parte de ella, pudiendo utilizar Javascript para crear todo tipo de lógica de apoyo.

Originalmente, a este sistema de realización de peticiones HTTP se le llamó AJAX, donde la X significa XML, el formato ligero de datos que más se utilizaba en aquel entonces. Actualmente, sobre todo en el mundo Javascript, se utiliza más el formato JSON, aunque por razones fonéticas evidentes (y probablemente evitar confundirlo con una risa) se sigue manteniendo el término AJAX.

Posteriormente, y debido a una evolución a mayor escala relacionada con este tema, se ha pasado de crear páginas de tipo MPA por defecto, a crear páginas de tipo SPA, mucho más frecuentes en entornos empresariales hoy en día. Hablaremos de ellos más adelante.



Métodos de petición AJAX

Existen varias formas de realizar peticiones HTTP mediante AJAX, pero las principales suelen ser [XMLHttpRequest](#) y [fetch](#) (nativas, incluidas en el navegador por defecto), además de librerías como [axios](#) o [superagent](#):

[XMLHttpRequest \(nativo\)](#): Se suele abreviar como XHR. El más antiguo, y también el más verbose. Nativo.

[fetch \(nativo\)](#): Nuevo sistema nativo de peticiones basado en promesas.

[Axios](#): Librería basada en promesas para realizar peticiones en Node o en navegadores.

[superagent](#): Librería para realizar peticiones HTTP tanto en Node como en navegadores.



XMLHttpRequest (XHR)

Se trata de la primera implementación que existió en [ECMAScript ES5](#) antes de surgir `fetch`, el estándar actual.

¿Qué es? **XMLHttpRequest (XHR)** Es un objeto especial de Javascript que permite realizar peticiones HTTP asíncronas (AJAX) de forma nativa desde Javascript.

Actualmente, es más frecuente utilizar `fetch`, puesto que es una API más actual y moderna que utiliza promesas y nos permite hacer lo mismo (o más) y escribir menos código.



Peticiones Asíncronas con fetch

Con el tiempo, el uso del objeto XMLHttpRequest (XHR) para realizar peticiones HTTP en JavaScript, aunque potente, puede resultar complicado y menos legible debido a la necesidad de manejar múltiples callbacks y estados. Para simplificar y modernizar el proceso, el método fetch fue introducido en la especificación de JavaScript, ofreciendo una forma más sencilla y legible de realizar peticiones asíncronas.

¿Qué es fetch?

El método fetch es una API de JavaScript que permite realizar peticiones HTTP de manera más fácil y con un enfoque basado en promesas. Esto hace que el código sea más limpio y fácil de entender, especialmente cuando se realizan múltiples operaciones asíncronas.



Ejemplo

```
// Realizar una petición GET usando fetch
fetch('https://jsonplaceholder.typicode.com/posts/1')
  .then(response => {
    // Verificar si la respuesta es exitosa
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json(); // Parsear la respuesta como JSON
  })
  .then(data => {
    console.log(data); // Trabajar con los datos recibidos
  })
  .catch(error => {
    console.error('There has been a problem with your fetch operation:', error);
  });
```

Cabeceras

Cabeceras HTTP

¿**Qué Son?** Las cabeceras HTTP son parte de la comunicación entre el navegador web (cliente) y el servidor. Se utilizan para enviar información adicional sobre la solicitud o la respuesta.

¿Por Qué Son Útiles para Principiantes?

1. **Configuración de Solicitudes:** Cuando envías una solicitud al servidor, puedes usar cabeceras para especificar detalles importantes, como el tipo de contenido que esperas (**Accept**) o el tipo de datos que estás enviando (**Content-Type**).
2. **Control de Caché:** Puedes usar cabeceras para controlar cómo se almacenan en caché las respuestas en el navegador (**Cache-Control**). Esto es útil para mejorar el rendimiento de la web.
3. **Seguridad:** Cabeceras como **Authorization** y **Cookie** son fundamentales para la autenticación y la gestión de sesiones, ayudando a mantener segura la comunicación entre el cliente y el servidor.
4. **Depuración:** Al trabajar con herramientas de desarrollo web, las cabeceras HTTP te permiten ver qué información está siendo enviada y recibida, lo que es útil para depurar problemas.

Ejemplo Sencillo: Cuando accedes a una página web, el navegador envía una solicitud al servidor. La cabecera **User-Agent** en la solicitud informa al servidor sobre el navegador y el sistema operativo que estás utilizando. Esto puede afectar cómo el servidor responde, como enviar un diseño específico para dispositivos móviles.




Cabeceras

Las **cabeceras** (o headers en inglés) en el contexto de HTTP y las peticiones web son pares de clave-valor que se envían junto con las solicitudes y respuestas entre un cliente (como un navegador web) y un servidor. Estas cabeceras proporcionan información adicional sobre la solicitud o la respuesta y pueden influir en cómo se procesa el contenido.

Tipos de Cabeceras

1. **Cabeceras de Solicitud (Request Headers):**
 - **Accept:** Indica el tipo de contenido que el cliente acepta recibir del servidor (por ejemplo, `text/html`, `application/json`).
 - **Authorization:** Lleva credenciales para autenticar al cliente en el servidor (por ejemplo, un token de acceso).
 - **Content-Type:** Especifica el tipo de contenido que se está enviando al servidor en el cuerpo de la solicitud (por ejemplo, `application/x-www-form-urlencoded`, `multipart/form-data`).
 - **User-Agent:** Identifica el cliente que está haciendo la solicitud (por ejemplo, el tipo de navegador o aplicación que está enviando la solicitud).
2. **Cabeceras de Respuesta (Response Headers):**
 - **Content-Type:** Indica el tipo de contenido que el servidor está enviando de vuelta (por ejemplo, `text/html`, `application/json`).
 - **Cache-Control:** Controla el comportamiento de la caché en el navegador o intermediarios (por ejemplo, `no-cache`, `max-age=3600`).
 - **Set-Cookie:** Envía cookies desde el servidor al cliente para almacenar en el navegador (por ejemplo, `session_id=12345`).
 - **Location:** Indica una URL para redirigir al cliente a otra página (por ejemplo, en respuestas de redirección).



A parte del método `.set()` podemos utilizar varios otros para trabajar con cabeceras, comprobar su existencia, obtener o cambiar los valores o incluso eliminarlos:

`.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`.



Respuesta de la petición HTTP

Cuando realizas una petición HTTP utilizando la función `fetch` en JavaScript, recibes un objeto de respuesta que contiene varios detalles útiles. Vamos a desglosar cómo trabajar con este objeto de respuesta y qué propiedades y métodos puedes utilizar para manejar la respuesta de manera efectiva.

El objeto `Response` de `fetch` proporciona una forma rica y flexible de manejar respuestas HTTP. Usando las propiedades y métodos descritos, puedes verificar el estado de la respuesta, leer datos, y trabajar con cabeceras y URLs. Esto hace que sea más fácil trabajar con peticiones HTTP y manejar tanto respuestas exitosas como errores de manera eficaz.

```
fetch('/robots.txt')
  .then((response) => {
    // Comprobar si la respuesta es exitosa
    if (response.ok) {
      // Leer el contenido de la respuesta como texto
      return response.text();
    } else {
      // Manejar errores si la respuesta no es exitosa
      throw new Error('Error en la solicitud: ' + response.statusText);
    }
  })
  .then((data) => {
    // Procesar el contenido de la respuesta
    console.log(data);
  })
  .catch((error) => {
    // Manejar errores que ocurrieron durante la petición
    console.error('Se produjo un error:', error);
  });
```



Propiedades del Objeto Response

1. `.status`:

- **Descripción:** Código de estado HTTP de la respuesta.
- **Rango:** 100 a 599.
- **Ejemplo:** `200` para éxito, `404` para no encontrado, `500` para error del servidor

```
fetch('/robots.txt')
  .then(response => {
    console.log('Código de estado:', response.status);
  });
```



.statusText

.statusText:

- **Descripción:** Texto descriptivo asociado con el código de estado.
- **Ejemplo:** "OK" para el código 200, "Not Found" para 404.

```
fetch('/robots.txt')  
  .then(response => {  
    console.log('Texto del estado:', response.statusText);  
  });
```



.ok

- **Descripción:** Propiedad booleana que indica si la respuesta fue exitosa (código de estado entre 200 y 299).
- **Ejemplo:** `true` para códigos como `200`, `false` para códigos como `404`

```
fetch('/robots.txt')
  .then(response => {
    if (response.ok) {
      console.log('La solicitud fue exitosa.');
```



.headers:

- **Descripción:** Objeto `Headers` que contiene las cabeceras de la respuesta HTTP.
- **Uso:** Puedes acceder a cabeceras específicas usando métodos como `.get(name)`.

```
fetch('/robots.txt')  
  .then(response => {  
    console.log('Cabeceras de la respuesta:', response.headers.get('Content-Type'));  
  });
```



.url

- **Descripción:** La URL completa de la petición realizada.
- **Uso:** Puede ser útil para confirmar la URL a la que se realizó la solicitud.

```
fetch('/robots.txt')  
  .then(response => {  
    console.log('URL de la solicitud:', response.url);  
  });
```



Métodos de procesamiento

La instancia response también tiene algunos métodos interesantes, la mayoría de ellos para procesar mediante una promesa los datos recibidos y facilitar el trabajo con ellos:

- `.text()`: Devuelve una promesa con el texto plano de la respuesta.
- `.json()`: Devuelve una promesa pero con un objeto json. Equivalente a usar `JSON.parse()`.
- `.blob()`: Devuelve una promesa pero con un objeto *Blob (binary large object)*.
- `.arrayBuffer()`: Devuelve una promesa pero con un objeto *ArrayBuffer* (buffer binario puro).
- `.formData()`: Devuelve una promesa, pero con un objeto *FormData* (datos de formulario).
- `.clone()`: Crea y devuelve un clon de la instancia en cuestión.
- `Response.error()`: Devuelve un nuevo objeto *Response* con un error de red asociado.
- `Response.redirect(url, code)`: Redirige a una url, opcionalmente con un *code* de error.



URL en JAVASCRIPT

Cuando tenemos que trabajar con una dirección web (URL) en Javascript, lo más habitual es utilizar un tipo de dato donde almacenamos dicha URL. En la mayoría de los casos, esto suele ser suficiente. Sin embargo, si necesitamos hacer ciertas operaciones con la URL donde tengamos que modificar o acceder a ciertas partes específicas de la URL, se podría complicar un poco.

Ejemplo:

Si necesitamos acceder a partes específicas de una URL, o incluso modificarlas, tenemos un object de tipo `URL` especial para estos casos, que será mucho más cómodo que trabajar con el `string` (y contemplar todos los posibles casos que podrían ocurrir). Su funcionamiento es el siguiente:

```
const url = new URL('https://www.google.com/');  
url.protocol; // "https:"  
url.hostname; // "www.google.com"  
url.pathname; // "/"
```

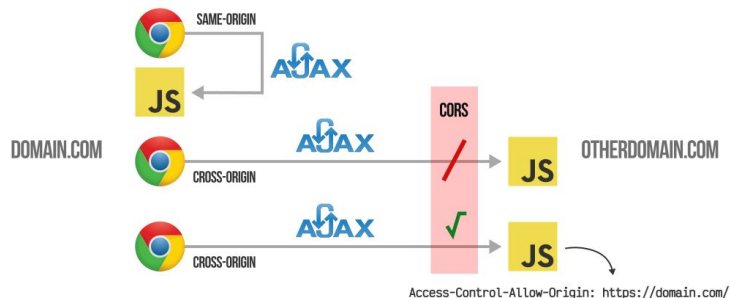

Política CORS

Habrás observado que estamos realizando peticiones relativas, es decir, al mismo dominio. Por defecto, en Javascript, las peticiones al mismo dominio de la web donde nos encontramos se pueden realizar sin ninguna restricción. Sin embargo, si intentamos realizarlas a otro dominio diferente, probablemente nos aparezca un error.

La política CORS (Cross-Origin Resource Sharing) es un mecanismo de seguridad implementado en los navegadores web para permitir o restringir el acceso a recursos web que están ubicados en un dominio diferente al del sitio web que realiza la solicitud. En otras palabras, CORS controla cómo los recursos solicitados desde un dominio pueden ser accedidos por otro dominio.

¿Qué es CORS?

CORS es una política de seguridad que se utiliza para permitir o restringir los recursos compartidos entre diferentes dominios. Los navegadores implementan esta política para prevenir que scripts maliciosos en una página web hagan solicitudes a otros dominios sin permiso, lo que podría llevar a problemas de seguridad como el robo de datos.



¿Cómo Funciona CORS?

Cuando un navegador realiza una solicitud HTTP a un dominio diferente (conocida como solicitud cross-origin), el servidor de destino debe permitir explícitamente esta solicitud. Esto se logra mediante el uso de cabeceras HTTP específicas que el servidor envía como respuesta.

Por defecto, los navegadores permiten enlazar hacia documentos situados en todo tipo de dominios si lo hacemos desde el HTML o desde Javascript utilizando la API DOM (que a su vez está construyendo un HTML). Sin embargo, no ocurre lo mismo cuando se trata de peticiones HTTP asíncronas mediante Javascript (AJAX), sea a través de XMLHttpRequest, de fetch o de librerías similares para el mismo propósito. Utilizando este tipo de peticiones asíncronas, los recursos situados en dominios diferentes al de la página actual no están permitidos por defecto. Es lo que se suele denominar protección de CORS. Su finalidad es dificultar la posibilidad de añadir recursos ajenos en un sitio determinado.

Ejemplo

Access to fetch at 'https://otherdomain.com/file.json' from origin 'https://domain.com/' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.