Asincronía

Fuente: lenguajejs.com / programacionymas.com

La **asincronía** es uno de los conceptos principales que rige el mundo de JavaScript. Cuando comenzamos a programar, normalmente realizamos tareas de forma **síncrona**, llevando a cabo tareas secuenciales que se ejecutan una detrás de otra, de modo que el orden o flujo del programa es sencillo y fácil de observar en el código:

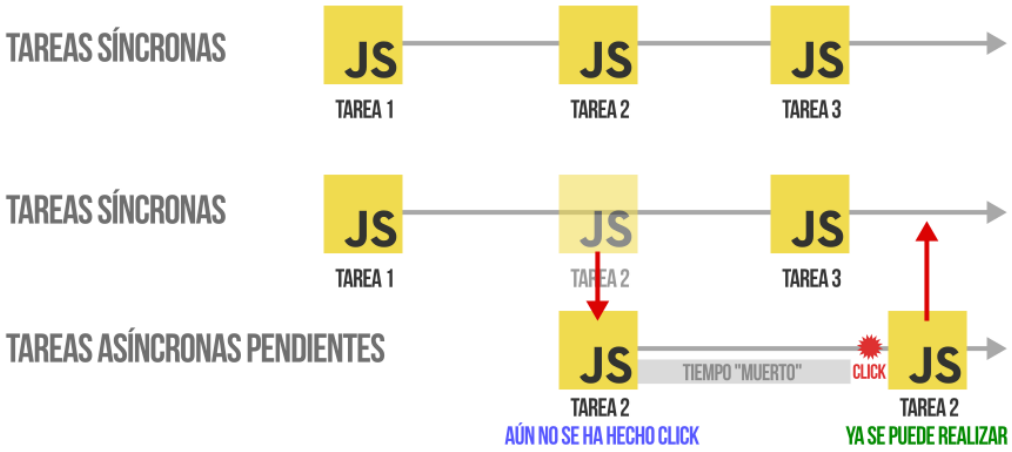
```
primera_funcion();    // Tarea 1: Se ejecuta primero
segunda_funcion();  // Tarea 2: Se ejecuta cuando termina primera_funcion()
tercera_funcion();    // Tarea 3: Se ejecuta cuando termina segunda_funcion()
```

Sin embargo, en el mundo de la programación, tarde o temprano necesitaremos realizar operaciones **asíncronas**, especialmente en ciertos lenguajes como JavaScript, 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.

Lenguaje no bloqueante

Cuando hablamos de JavaScript, habitualmente nos referimos a él como un lenguaje **no bloqueante**. Con esto queremos decir que las tareas que realizamos no se quedan bloqueadas esperando ser finalizadas, y por consiguiente, evitando proseguir con el resto de tareas.

Imaginemos que la **segunda_funcion()** del ejemplo anterior realiza una tarea que depende de otro factor, como por ejemplo un click de ratón del usuario. Si hablasemos de un **lenguaje bloqueante**, hasta que el usuario no haga click, JavaScript no seguiría ejecutando las demás funciones, sino que se quedaría bloqueado esperando a que se terminase esa segunda tarea:



Pero como JavaScript es un **lenguaje no bloqueante**, lo que hará es mover esa tarea a una lista de **tareas pendientes** a las que irá «*prestando atención*» a medida que lo necesite, pudiendo continuar y retomar el resto de tareas a continuación de la segunda.

¿Qué es la asincronía?

Pero esto no es todo. Ten en cuenta que pueden existir **múltiples** tareas asíncronas, dichas tareas puede que terminen realizándose correctamente (*o puede que no*) y ciertas tareas pueden depender de otras, por lo que deben respetar un cierto orden. Además, es muy habitual que no sepamos previamente **cuánto tiempo** va a tardar en terminar una tarea, por lo que necesitamos un mecanismo para controlar todos estos factores.

¿Cómo gestionar la asincronía?

Teniendo en cuenta el punto anterior, debemos aprender a buscar mecanismos para dejar claro en nuestro código JavaScript, que ciertas tareas tienen que procesarse de forma asíncrona para quedarse a la espera, y otras deben ejecutarse de forma síncrona.

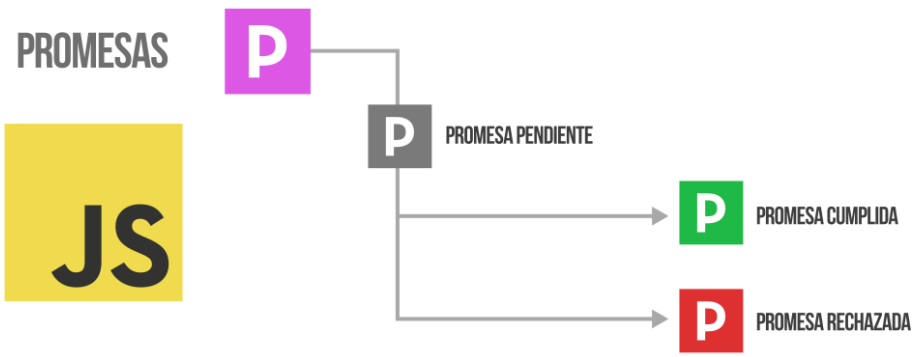
En JavaScript existen varias formas de gestionar la **asincronía**, donde quizás las más populares son las siguientes:

Método	Descripción
Mediante **callbacks**	Probablemente, la forma más clásica de gestionar la asincronía en JavaScript.
Mediante **promesas**	Una forma más moderna y actual de gestionar la asincronía.
Mediante **async**/**await**	Seguimos con promesas, pero con **async**/**await** añadimos más azúcar sintáctico.

Promesas

Las **promesas** son un concepto para resolver el problema de asincronía de una forma mucho más elegante y práctica.

Como su propio nombre indica, una **promesa** es algo que, en principio pensamos que se cumplirá, pero en el futuro pueden ocurrir varias cosas:



- La promesa **se cumple** (*promesa resuelta*)
- La promesa **no se cumple** (*promesa se rechaza*)
- La promesa se queda en un **estado incierto** indefinidamente (*promesa pendiente*)

Básicamente, las Promesas son similares a los Eventos, con las siguientes diferencias:

- Una promesa sólo puede tener éxito o fracasar una única vez. No puede tener éxito o fallar por una 2da vez, ni cambiar de éxito a fallo posteriormente, o viceversa.
- Si una promesa ha sido exitosa o ha fallado, y más adelante (recién) registramos un callback de success o failure, la función de callback correspondiente será llamada (incluso si el evento tuvo lugar antes).

Esto resulta muy útil para operaciones asíncronas, porque más allá de capturar el momento exacto en que ocurre algo, nos enfocamos en reaccionar ante lo ocurrido.

Terminología asociada a las Promesas

Tenemos muchos términos relacionados a lo que son Promesas en Javascript. A continuación veamos lo más básico.

Una promesa puede presentar los siguientes estados:

- **fulfilled** - La acción relacionada a la promesa se llevó a cabo con éxito
- **rejected** - La acción relacionada a la promesa falló
- **pending** - Aún no se ha determinado si la promesa fue fulfilled o rejected
- **settled** - Ya se ha determinado si la promesa fue fulfilled o rejected

También se suele usar el término **thenable**, para indicar que un objeto tiene disponible un método "then" (y que por tanto está relacionado con Promesas).

Promesas en JavaScript

Las **promesas** en JavaScript se representan a través de un **object**, y cada **promesa** estará en un estado concreto: **pendiente**, **aceptada** o **rechazada**. Además, cada **promesa** tiene los siguientes métodos, que podremos utilizar para utilizarla:

Métodos	Descripción
.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.

Cómo crear una Promesa en JS

Un objeto Promise representa un valor, que no se conoce necesariamente al momento de crear la promesa.

Esta representación nos permite realizar acciones, con base en el valor de éxito devuelto, o la razón de fallo.

Es decir, los métodos asíncronos producen valores que aún no están disponibles. Pero la idea ahora es que, en vez de esperar y devolver el valor final, tales métodos devuelven un objeto Promise (que nos proveerá del valor resultante en el futuro).

Hoy en día, muchas bibliotecas JS se están actualizando para hacer uso de Promesas, en vez de simples funciones callback.

Nosotros también podemos crear nuestras promesas, basados en esta sintaxis:

```
new Promise(function(resolve, reject) { ... });
```

Este constructor es usado principalmente para envolver funciones que no soportan el uso de Promesas.

- El constructor espera una función como parámetro. A esta función se le conoce como **executor**.
- Esta función executor recibirá 2 argumentos: **resolve** y **reject**.
- La función executor es ejecutada inmediatamente al implementar el objeto Promise, recibiendo las funciones resolve y reject para su uso correspondiente. Esta función executor es llamada incluso antes que el constructor Promise devuelva el objeto creado.
- Las funciones resolve y reject, al ser llamadas, "resuelven" o "rechazan" la promesa. Es decir, modifican el estado de la promesa (como hemos visto antes, inicialmente es pending, pero posteriormente puede ser fulfilled o rejected).
- Normalmente el executor inicia alguna operación asíncrona, y una vez que ésta se completa, llama a la función resolve para resolver la promesa o bien reject si ocurrió algo inesperado.
- Si la función executor lanza algún error, la promesa también es rejected.
- El valor devuelto por la función executor es ignorado.

```
let promise = new Promise(function(resolve, reject) {
  function sayHello() {
    resolve('Hello World!');
  }
  setTimeout(sayHello, 3000);
});
console.log(promise);
```

Lo que ha de ocurrir es lo siguiente:

- Se ejecuta la función executor y se crea nuestro objeto Promise.
- Se llama al método then, expresando qué es lo que queremos hacer con el valor que devolverá la promesa.
- Se imprime por consola [object Promise].

Ejemplo con .then(), .catch() y .finally()

```
let promesa = new Promise(function(resolve,reject){
	if(true){
		resolve(`Funcionó!`);
	}
	else{
		reject(`Hay un error`);
	}
});

promesa.then(function(respuesta){
	console.log(`Respuesta: ${respuesta}`);
})
.catch(function(error){
	console.log(`Error: ${error}`);
})
.finally(function(){
	console.log(`Esto se ejecuta siempre`);
});
```

API de las promesas

El objeto **Promise** de Javascript tiene varios métodos que podemos utilizar en nuestro código. Todos devuelven una promesa y son los que veremos en la siguiente tabla:

Métodos	Descripción
Promise.all([array]list)	Acepta sólo si **todas** las promesas del **array** se cumplen.

Promise.allSettled([array]list)	Acepta sólo si **todas** las promesas del **array** se cumplen o rechazan.
Promise.any([array]list)	Acepta con el valor de la primera promesa del **array** que se cumpla.
Promise.race([array]list)	Acepta o rechaza dependiendo de la primera promesa que se procese.
Promise.resolve(value)	Devuelve un valor envuelto en una promesa que se cumple directamente.
Promise.reject(value)	Devuelve un valor envuelto en una promesa que se rechaza directamente.

En los siguientes ejemplos, vamos a utilizar la función **fetch()** para realizar varias peticiones y descargar varios archivos diferentes que necesitaremos para nuestras tareas.

Promise.all()

El método **Promises.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.

A **Promise.all()** le pasamos un **array** con las promesas individuales. Cuando **todas y cada una** de esas promesas se cumplan favorablemente, **entonces** se ejecutará la función callback de su **.then()**. En el caso de que alguna se rechace, no se llegará a ejecutar.

Promise.allSettled()

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. });

Esta promesa nos devuelve un campo **status** donde nos indica si cada promesa individual ha sido cumplida o rechazada, y un campo **value** con los valores devueltos por la promesa.

Promise.any()

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.any() devolverá una respuesta de la primera promesa cumplida.

Promise.race()

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.

De forma muy similar a la anterior, **Promise.race()** devolverá la promesa que se resuelva primero, ya sea cumpliéndose o rechazándose.

La palabra clave async

En **ES2017** se introducen las palabras clave **async**/**await**, que no son más que una forma de **azúcar sintáctico** para gestionar las promesas de una forma más sencilla. Con **async**/**await** seguimos utilizando promesas, pero abandonamos el modelo de encadenamiento de **.then()** para utilizar uno en el que trabajamos de forma más tradicional.

En primer lugar, tenemos la palabra clave **async**. Esta palabra clave se colocará previamente a **function**, para definirla así como una **función asíncrona**, el resto de la función no cambia:

```
async function funcion_asincrona() {
  return 42;
}
```

En el caso de que utilicemos **arrow function**, se definiría como vemos a continuación, colocando el **async** justo antes de los parámetros de la arrow function:

```
const funcion_asincrona = async () => 42;
```

Al ejecutar la función veremos que ya nos devuelve una que ha sido cumplida, con el valor devuelto en la función (*en este caso, 42*). De hecho, podríamos utilizar un **.then()** para manejar la promesa:

```
funcion_asincrona().then(function(value){
  console.log("El resultado devuelto es: ", value);
});
```

Sin embargo, veremos que lo que se suele hacer junto a **async** es utilizar la palabra clave **await**, que es donde reside lo interesante de utilizar este enfoque.

La palabra clave await

Cualquier función definida con **async**, o lo que es lo mismo, cualquier **Promise** puede utilizarse junto a la palabra clave **await** para manejarla. Lo que hace **await** es esperar a que se resuelva la promesa, mientras permite continuar ejecutando otras tareas que puedan realizarse:

```
const funcion_asincrona = async function(){return 42;}

const value = funcion_asincrona();             // Promise { <fulfilled>: 42 }
const asyncValue = await funcion_asincrona();  // 42
```

Observa que en el caso de **value**, que se ejecuta sin **await**, lo que obtenemos es el valor devuelto por la función, pero «envuelto» en una promesa que deberá utilizarse con **.then()** para manejarse. Sin embargo, en **asyncValue** estamos obteniendo un tipo de dato **Number** , guardando el valor directamente ya procesado, ya que **await** espera a que se resuelva la promesa de forma asíncrona y guarda el valor.

> Esto hace que la forma de trabajar con **async**/**await**, aunque se sigue trabajando exactamente igual con promesas, sea mucho más fácil y trivial para usuarios que no estén acostumbrados a las promesas y a la asincronía en general, ya que el código «parece» síncrono.

> Recuerda que en el caso de querer controlar errores o promesas rechazadas, siempre podrás utilizar bloques **try**/**catch**.

¿Qué es una petición 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, ya sea un fichero **.html**, una imagen, una tipografía, un archivo **.js**, 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.

HTTP define una gran cantidad de métodos que son utilizados para diferentes circunstancias:

- **GET:** Es utilizado únicamente para **consultar información** al servidor, muy parecido a realizar un SELECT a la base de datos.
- **POST:** Es utilizado para solicitar la **creación de un nuevo registro**, es decir, algo que no existía previamente, es equivalente a realizar un INSERT en una base de datos.
- **PUT**: Se utiliza para **actualizar por completo un registro existente**, es decir, es parecido a realizar un UPDATE en la base de datos.
- **PATCH**: Este método es similar al método PUT, pues permite actualizar un registro existente, sin embargo, este se utiliza cuando **es necesario actualizar solo un fragmento del registro** y no en su totalidad, es equivalente a realizar un UPDATE a la base de datos.
- **DELETE**: Este método se utiliza para **eliminar un registro existente**, es similar a DELETE en la base de datos.
- **HEAD**: Este método se utiliza para **obtener información sobre un determinado recurso** sin retornar el registro. Este método se utiliza a menudo para probar la validez de los enlaces de hipertexto, la accesibilidad y las modificaciones recientes.

Peticiones HTTP mediante AJAX

Con el tiempo, aparece una nueva modalidad de realizar peticiones, denominada **AJAX** (*Asynchronous Javascript and XML*). 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 evitar confundirlo con una risa*) se sigue manteniendo el término **AJAX**.

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 liberías como **axios** o **superagent**:

Peticiones HTTP con fetch

Fetch es el nombre de una nueva API para Javascript con la cual podemos realizar peticiones HTTP asíncronas utilizando promesas y de forma que el código sea un poco más sencillo. La forma de realizar una petición es muy sencilla, básicamente se trata de llamar a **fetch** y pasarle por parámetro la URL de la petición a realizar:

```
// Realizamos la petición y guardamos la promesa
const request = fetch("./robots.txt");

// Si es resuelta, entonces...
request.then(function(response) { ... });
```

El **fetch()** devolverá una promesa que será aceptada cuando reciba una respuesta y sólo será rechazada si hay un fallo de red o si por alguna razón no se pudo completar la petición. El modo más habitual de manejar las promesas es utilizando **.then()**. Esto se suele reescribir de la siguiente forma, que queda mucho más simple:

```
fetch("/robots.txt")
  .then(function(response) {
    /** Código que procesa la respuesta **/
  });
```

Al método **.then()** se le pasa una función callback donde su parámetro **response** es el objeto de respuesta de la petición que hemos realizado. En su interior realizaremos la lógica

que queramos hacer con la respuesta a nuestra petición. A la función **fetch(url, options)** se le pasa por parámetro la **url** de la petición y, de forma opcional, un objeto **options** con opciones de la petición HTTP.

Vamos a examinar un código donde veamos un poco mejor como hacer la petición con **fetch**:

```
// Opciones de la petición (valores por defecto)
const options = {
  method: "GET"
};

// Petición HTTP
fetch("/robots.txt", options)
  .then(function(response){
          return response.text();
  })
  .then(function(data){
    /** Procesar los datos **/
  });
```

Vamos a centrarnos ahora en el parámetro opcional **options** de la petición HTTP. En este objeto podemos definir varios detalles:

Campo	Descripción
method	Método HTTP de la petición. Por defecto, **GET**. Otras opciones: **HEAD**, **POST**, etc...
body	Cuerpo de la petición HTTP (El cuerpo del mensaje HTTP son los bytes de datos transmitidos en un mensaje de transacción HTTP inmediatamente después de los encabezados, si los hay). Puede ser de varios tipos: **String**, **FormData**, **Blob**, etc.

headers	Cabeceras o encabezados HTTP (parámetros que se envían en una petición o respuesta HTTP al cliente o al servidor para proporcionar información esencial sobre la transacción en curso). Por defecto, **{}**.
credentials	Modo de credenciales. Por defecto, **omit**. Otras opciones: **same-origin** e **include**.

Lo primero, y más habitual, suele ser indicar el método HTTP a realizar en la petición. Por defecto, se realizará un **GET**, pero podemos cambiarlos a **HEAD**, **POST**, **PUT** o cualquier otro tipo de método. En segundo lugar, podemos indicar objetos para enviar en el **body** de la petición, así como modificar las cabeceras en el campo **headers**:

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

Por último, el campo **credentials** permite modificar el modo en el que se realiza la petición. Por defecto, el valor **omit** hace que no se incluyan credenciales en la petición, pero es posible indicar los valores **same-origin**, que incluye las credenciales si estamos sobre el mismo dominio, o **include** que incluye las credenciales incluso en peticiones a otros dominios.

Recuerda que estamos realizando peticiones relativas al **mismo dominio**. En el caso de realizar peticiones a dominios diferentes obtendremos un problema de CORS (*Cross-Origin Resource Sharing*) similar al siguiente:

> 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.

Cabeceras (Headers)

Aunque en el ejemplo anterior hemos creado las cabeceras como un objeto genérico de Javascript, es posible crear un objeto **Headers** con el que trabajar:

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

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

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

Respuesta de la petición HTTP

Si volvemos a nuestro ejemplo de la petición con **fetch**, observaremos que en el primer **.then()** tenemos un objeto **response**. Se trata de la respuesta que nos llega del servidor web al momento de recibir nuestra petición:

```
// Petición HTTP
fetch("/robots.txt", options)
  .then(function(response){
            return response.text();
})
  .then(function(data){
    /** Procesar los datos **/
  });
```

El objeto **response** tiene una serie de propiedades y métodos que pueden resultarnos útiles al implementar nuestro código.

Propiedad	Descripción
.status	Código de error HTTP de la respuesta (100-599).
.statusText	Texto representativo del código de error HTTP anterior.
.ok	Devuelve **true** si el código HTTP es **200** (o empieza por **2**).

.headers	Cabeceras de la respuesta.
.url	URL de la petición HTTP.

Las propiedades **.status** y **statusText** nos devuelven el **código de error HTTP** de la respuesta en formato numérico y cadena de texto respectivamente.

Tenemos una propiedad **.ok** que nos devuelve **true** si el código de error de la respuesta es un valor del rango **2xx**, es decir, que todo ha ido correctamente. Así pues, tenemos una forma práctica y sencilla de comprobar si todo ha ido bien al realizar la petición:

```
fetch("/robots.txt")
  .then(function(response) {
    if (response.ok)
      return response.text();
  })
```

Por último, tenemos la propiedad **.headers** que nos devuelve las cabeceras de la respuesta y la propiedad **.url** que nos devuelve la URL completa de la petición que hemos realizado.

Métodos de procesamiento

Por otra parte, 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:

Método	Descripción
.text()	Devuelve una promesa con el texto plano de la respuesta.
.json()	Idem, pero con un objeto **json**. Equivalente a usar **JSON.parse()**.
.blob()	Idem, pero con un objeto **Blob** (binary large object).
.arrayBuffer()	Idem, pero con un objeto **ArrayBuffer** (buffer binario puro).
.formData()	Idem, 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.

Observa que en los ejemplos anteriores hemos utilizado **response.text()**. Este método indica que queremos procesar la respuesta como datos textuales, por lo que dicho método devolverá un **string** con los datos en texto plano, facilitando trabajar con ellos de forma manual:

```
fetch("/robots.txt")
  .then(function(response){return response.text();})
  .then(function(data){console.log(data)});
```

Observa que en este fragmento de código, tras procesar la respuesta con **response.text()**, devolvemos una con el contenido en texto plano. Esta se procesa en el segundo **.then()**, donde gestionamos dicho contenido almacenado en **data**.

Ten en cuenta que tenemos varios métodos similares para procesar las respuestas. Por ejemplo, el caso anterior utilizando el método **response.json()** en lugar de **response.text()** sería equivalente al siguiente fragmento:

```
fetch("/contents.json")
  .then(function(response){ return response.text();})
  .then(function(data){
    const json = JSON.parse(data);
    console.log(json);
  });
```

Como se puede ver, con **response.json()** nos ahorraríamos tener que hacer el **JSON.parse()** de forma manual, por lo que el código es algo más directo.

Ejemplo utilizando promesas

Lo que vemos a continuación sería un ejemplo un poco más completo de todo lo que hemos visto hasta ahora:

- Comprobamos que la petición es correcta con **response.ok**
- Utilizamos **response.text()** para procesarla
- En el caso de producirse algún error, lanzamos excepción con el código de error
- Procesamos los datos y los mostramos en la consola
- En el caso de que la sea rechazada, capturamos el error con el **catch**

```
// Petición HTTP
fetch("/robots.txt")
  .then(function(response){
    if (response.ok)
      return response.text()
    else
      throw new Error(response.status);
  })
  .then(function(data){
    console.log("Datos: " + data);
  })
  .catch(function(err){
    console.error("ERROR: ", err.message)
  });
```

Sin embargo, aunque es bastante común trabajar con promesas utilizando **.then()**, también podemos hacer uso de **async/await** para manejar promesas, de una forma un poco más directa.

Ejemplo utilizando Async/await

Utilizar **async/await** no es más que lo que se denomina **azúcar sintáctico**, es decir, utilizar algo visualmente más agradable, pero que por debajo realiza la misma tarea. Para ello, lo que debemos tener siempre presente es que un **await** sólo se puede ejecutar si esta dentro de una función definida como **async**.

En este caso, lo que hacemos es lo siguiente:

- Creamos una función **request(url)** que definimos con **async**
- Llamamos a **fetch** utilizando **await** para esperar y resolver la promesa
- Comprobamos si todo ha ido bien usando **response.ok**
- Llamamos a **response.text()** utilizando **await** y devolvemos el resultado

```
const request = async function(url){
  const response = await fetch(url);
  if (!response.ok)
    throw new Error("WARN", response.status);
  const data = await response.text();
  return data;
}

const resultOk = await request("/robots.txt");
const resultError = await request("/nonExistentFile.txt");
```

Una vez hecho esto, podemos llamar a nuestra función **request** y almacenar el resultado, usando nuevamente **await**. Al final, utilizar **.then()** o **async/await** es una cuestión de gustos y puedes utilizar el que más te guste.

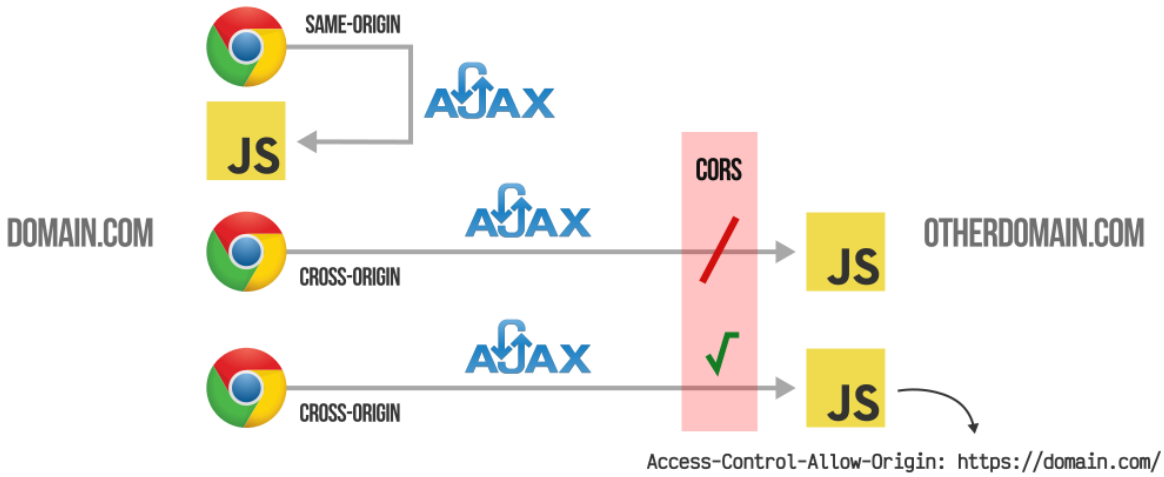
¿Qué es CORS?

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

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

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.



Si intentamos realizar una petición asíncrona hacia otro dominio diferente, probablemente obtendremos un error de CORS similar al siguiente:

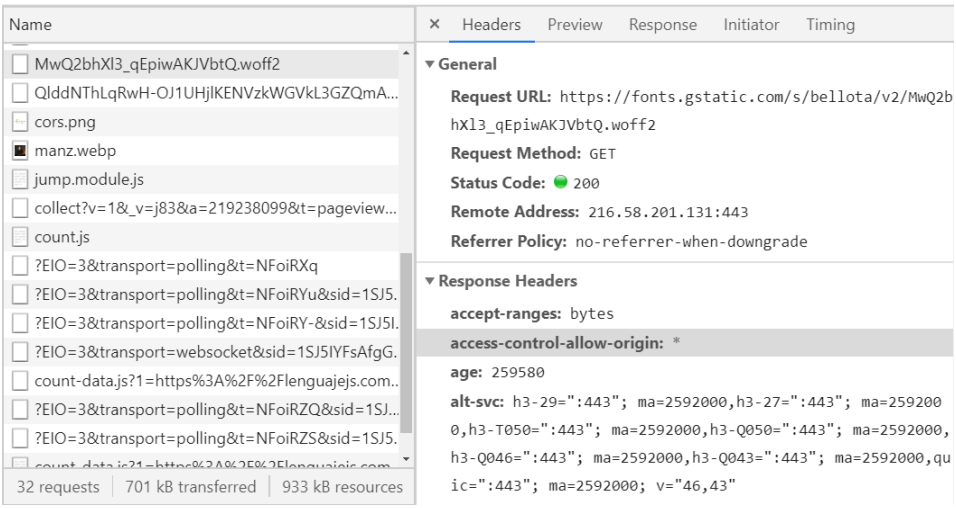
> Access to fetch at 'https://otherdomain.com/file.json' from origin 'https://domain.com/' has been blocked by **CORS policy**: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource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

Access-Control-Allow-Origin

Como hemos comentado, las peticiones HTTP asíncronas de origen cruzado no están permitidas, pero existen formas de permitirlas. La más básica, probablemente, sea la de incluir una cabecera **Access-Control-Allow-Origin** en la respuesta de la petición, donde debe indicarse el dominio al que se le quiere dar permiso:

```
Access-Control-Allow-Origin: https://domain.com/
```

De esta forma, el navegador comprobará dichas cabeceras y si coinciden con el dominio de origen que realizó la petición, esta se permitirá. En el ejemplo anterior, la cabecera tiene el valor **https://domain.com/**, pero en algunos casos puede interesar indicar el valor *.



> El asterisco * indica que se permiten peticiones de origen cruzado **a cualquier dominio**, algo que puede ser interesante cuando se tienen API públicas a las que quieres permitir el acceso al público en general, casos como los de Google Fonts o JSDelivr, por citar un ejemplo.

Estas cabeceras puedes verlas fácilmente accediendo a la pestaña **Network** del **Developer Tools** del navegador. En esta sección te aparecerá una lista de peticiones realizadas por el navegador en la página actual. Si seleccionamos una petición y accedemos al apartado de cabeceras (*Headers*), podremos examinar si existe la cabecera **Access-Control-Allow-Origin**:

CORS en entornos de desarrollo

Otra opción sencilla y rápida para no tener que lidiar con CORS **temporalmente** es la de instalar la **extensión Allow CORS**, disponible tanto Allow CORS para Chrome como Allow CORS para Firefox. Esta extensión deshabilita la **política CORS** mientras está instalada y activada. Esta elección es equivalente a que todas las respuestas a las peticiones asíncronas realizadas tengan la mencionada cabecera con el valor *. Obviamente, es importante recalcar que es una opción que **sólo nos funcionará en nuestro equipo y navegador**, pero puede ser muy práctica para simplificar el trabajo en desarrollo.

Desactivar CORS en JavaScript: Agregar dentro de la propiedad **headers:**

```
'Access-Control-Allow-Origin' : 'https://sitio.com'
```