

JAVASCRIPT FRONTEND

AVANZADO

CLASE 4 : AJAX

ASYNC vs SYNC

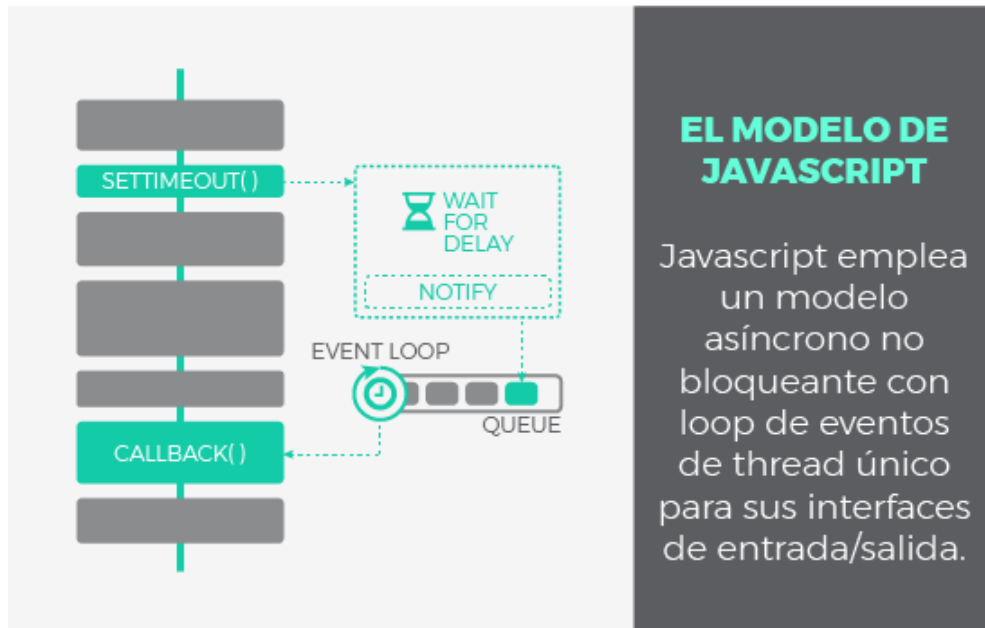
La asincronía es uno de los pilares fundamentales de Javascript. El objetivo de esta guía es profundizar en las piezas y elementos que la hacen posible. Teniendo claro estos conceptos, podrás ponerlos en práctica en tu código y escribir mejores aplicaciones. Podemos tener operaciones I/O de tipo:

- **Síncronas y Bloqueantes** : Toda la operación se hace de una vez, bloqueando el flujo de ejecución:
El thread es bloqueado mientras espera.
La respuesta se procesa inmediatamente después de terminar la operación.
- **Síncronas y No-Bloqueante** : Similar a la anterior pero usando alguna técnica de polling para evitar el bloqueo en la primera fase:
La llamada devuelve inmediatamente, el thread no se bloquea. Se necesitarán sucesivos intentos hasta completar la operación.
La respuesta se procesa inmediatamente después de terminar la operación.
- **Asíncronas y No-Bloqueantes**: a petición devuelve inmediatamente para evitar el bloqueo.
Se envía una notificación una vez que la operación se ha completado. Es entonces cuando la función que procesa la respuesta (callback) se encola para ser ejecutada en algún momento en nuestra aplicación.

Javascript fue diseñado para ser ejecutado en navegadores, trabajar con peticiones sobre la red y procesar las interacciones de usuario, al tiempo que se mantiene una interfaz fluida.

Ser bloqueante o síncrono no ayudaría a conseguir estos objetivos, es por ello que Javascript ha evolucionado intencionadamente pensando en operaciones de tipo I/O. Por esta razón:

"Javascript utiliza un modelo asíncrono y no bloqueante, con un loop de eventos implementado con un único thread para sus interfaces de entrada/salida"



BUCLE DE EVENTOS

¿Cómo se ejecuta un programa en Javascript? ¿Como gestiona nuestra aplicación de forma concurrente las respuestas a las llamadas asíncronas? Eso es exactamente lo que el modelo basado en un loop de eventos viene a responder:



Call Stack

Traducido, pila de llamadas, se encarga de albergar las instrucciones que deben ejecutarse. Nos indica en qué punto del programa estamos, por donde vamos. Cada llamada a función de nuestra aplicación, entra a la pila generando un nuevo frame (bloque de memoria reservada para los argumentos y variables locales de dicha función). Por tanto, cuando se llama a una función, su frame es insertado arriba en la pila, cuando una función se ha completado y devuelve, su frame se saca de la pila también por arriba. El funcionamiento es LIFO: last in, first out. De este modo, las llamadas a función que están dentro de otra función contenedora son apiladas encima y serán atendidas primero.

Una de las implicaciones más relevantes de este bucle de eventos es que los callbacks no serán despachados tan pronto como sean encolados, sino que deben esperar su turno. Este tiempo de espera dependerá del número de mensajes pendientes de procesar (por delante en la cola) así como del tiempo que se tardará en cada uno de ellos. Aunque pueda parecer obvio, esto explica la razón por la cual la finalización de una operación asíncrona no puede predecirse con seguridad, sino que se atiende en modo best effort.

El loop de eventos no está libre de problemas, y podrían darse situaciones comprometidas en los siguientes casos:

- La pila de llamadas no se vacía ya que nuestra aplicación hace uso intensivo de ella. No habrá tick en el bucle de eventos y por tanto los mensajes no se procesan.
- El flujo de mensajes que se van encolando es mayor que el de mensajes procesados. Demasiados eventos a la vez.
- Un callback requiere procesamiento intensivo y acapara la pila. De nuevo bloqueamos los ticks del bucle de eventos y el resto de mensajes no se despachan.

Lo más probable es que un cuello de botella se produzca como consecuencia de una mezcla de factores. En cualquier caso, acabarían retrasando el flujo de ejecución. Y por tanto retrasando el renderizado, el procesamiento de eventos, etc. La experiencia de usuario se degradará y la aplicación dejaría de responder de forma fluida. Para evitar esta situación, recuerda siempre mantener los callbacks lo más ligeros posible. En general, evita código que acapara la CPU y permite que el loop de eventos se ejecute a buen ritmo.

PATRONES ASINCRÓNICOS

Los callbacks son la pieza clave para que Javascript pueda funcionar de forma asíncrona. De hecho, el resto de patrones asíncronos en Javascript está basado en callbacks de un

modo u otro, simplemente añaden azúcar sintáctico para trabajar con ellos más cómodamente.

Un callback no es más que una función que se pasa como argumento de otra función, y que será invocada para completar algún tipo de acción. En nuestro contexto asíncrono, un callback representa el '¿Qué quieres hacer una vez que tu operación asíncrona termine?'. Por tanto, es el trozo de código que será ejecutado una vez que una operación asíncrona notifique que ha terminado. Esta ejecución se hará en algún momento futuro, gracias al mecanismo que implementa el bucle de eventos.

Fíjate en el siguiente ejemplo sencillo utilizando un callback:

```
setTimeout(function(){  
  console.log("Hola Mundo con retraso!");  
}, 1000)
```

setTimeout es una función asíncrona que programa la ejecución de un callback una vez ha transcurrido, como mínimo, una determinada cantidad de tiempo (1 segundo en el ejemplo anterior). A tal fin, dispara un timer en un contexto externo y registra el callback para ser ejecutado una vez que el timer termine. En resumen, retrasa una ejecución, como mínimo, la cantidad especificada de tiempo.

Es importante comprender que, incluso si configuramos el retraso como 0ms, no significa que el callback vaya a ejecutarse inmediatamente. Atento al siguiente ejemplo:

```
setTimeout(function(){  
  console.log("Esto debería aparecer primero");  
}, 0);  
console.log("Sorpresa!");
```

Recuerda, un callback que se añade al loop de eventos debe esperar su turno. En nuestro ejemplo, el callback del setTimeout debe esperar el primer tick. Sin embargo, la pila esta ocupada procesando la línea console.log("Sorpresa!"). El callback se despachará una vez la pila quede vacía, en la práctica, cuando Sorpresa! haya sido logueado.⁴

AJAX

Qué es y para qué nos sirve AJAX?

JavaScript Asíncrono y XML (AJAX) no es una tecnología por sí misma, es un término que describe un nuevo modo de utilizar conjuntamente varias tecnologías existentes. Esto incluye: **HTML** o **XHTML**, **CSS**, **JavaScript**, **DOM**, **XML**, **XSLT**, y el objeto **XMLHttpRequest**. Cuando estas tecnologías se combinan en un modelo AJAX, es posible lograr aplicaciones web capaces de actualizarse continuamente sin tener que volver a cargar la página completa. Esto crea aplicaciones más rápidas y con mejor respuesta a las acciones del usuario. ¹

XMLHttpRequest API

XMLHttpRequest es un objeto JavaScript que fue diseñado por Microsoft y adoptado por Mozilla, Apple y Google. Actualmente es un estándar de la W3C. Proporciona una forma fácil de obtener información de una URL sin tener que recargar la página completa

A pesar de su nombre, **XMLHttpRequest** puede ser usado para recibir cualquier tipo de dato, no sólo XML, y admite otros formatos además de HTTP (incluyendo file y ftp).

Para crear una instancia de **XMLHttpRequest**, debes hacer lo siguiente ³:

```
var xhr = new XMLHttpRequest();
```

Tengamos en cuenta que versiones viejas de Internet Explorer pueden tener una implementación anterior en su JScript por lo que si queremos tener compatibilidad para esos navegadores deberíamos hacer ² :

```
var xhr ;
if (window.XMLHttpRequest) { // Mozilla, Safari, ...
    xhr = new XMLHttpRequest();
} else if (window.ActiveXObject) { // IE
    xhr = new ActiveXObject("Microsoft.XMLHTTP");
}
```

El objeto que obtengamos a partir de la inicialización de la API XHR contiene propiedades y métodos que nos permiten establecer comunicación asincrónica con otra máquina a través del protocolo HTTP.

ReadyState

Como podemos observar si mostramos el objeto que acabamos de crear en la consola, notamos que todas sus propiedades se encuentran vacías en null, undefined, 0(cero) ó ""(string vacío). Esto es porque aún no lo configuramos para que pueda realmente enviar nuestra petición.

De todas las propiedades que se muestran, la que sí nos está dando información es la propiedad `XHR.readyState`. La misma nos indica el estado en el que se encuentra una solicitud y la misma puede tomar los siguientes valores :

ESTADO	DESCRIPCIÓN
0 - UNSENT	Objeto Inicializado
1 - OPENED	Objeto configurado
2 - HEADERS_RECEIVED	El objeto ya se envió y el status de los headers del servidor ya volvieron
3 - LOADING	Descargando. La propiedad <code>responseText</code> mantiene datos parciales
4 - DONE	La operación se completó no necesariamente exitosa

Open

El método `.open()` nos permite configurar una solicitud saliente ya inicializada o reconfigurar una que ya se encontraba configurada.

```
XMLHttpRequest.open(String metodo,String url[, Boolean async])
```

Recibe como parámetros obligatorios el método HTTP por el cual vamos a enviar la solicitud y la URL a la cual la vamos a enviar. Como último parámetro opcional recibe un booleano que determina si la solicitud va a ser sincrónica o asincrónica.

Configurar un objeto XHR no implica que ya lo hayamos enviado, sino que está listo para poder enviarse a futuro.

ReadyStateChange

Todos los objetos que extiendan de la api XHR tienen este evento que se dispara cada vez que la propiedad `readyState` cambia. Con él podemos tener control total sobre cada uno de los estados del pedido :

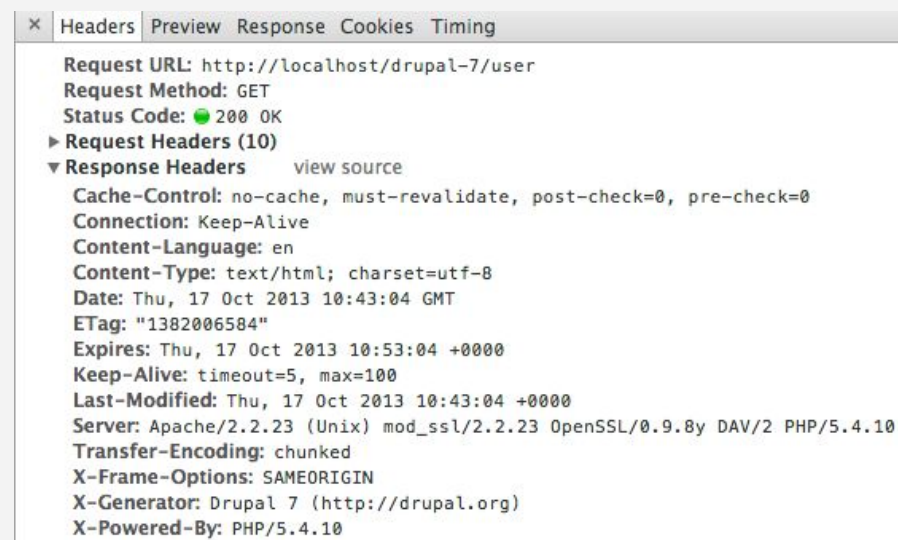
```
var xhr = new XMLHttpRequest()
xhr.addEventListener("readystatechange",function(){
    console.log(xhr.readyState)
})
xhr.open("GET","url.com")
```

Como podemos observar, vamos a tener en consola el mensaje "1" ya que en la primer línea de código teníamos el estado 0(cero) pero luego de asignarle el evento lo configuramos, por lo tanto el mismo cambió su estado a 1(un). Eventualmente nos vamos a enterar cuando otras cosas pasen durante el transcurso de nuestro pedido ya que el mismo evento va a ejecutar su callback cada vez que el objeto XHR cambie de estado.

Este es un buen lugar para realizar comprobaciones de peso y tipo de recurso solicitado, en el caso de que hayamos solicitado alguno, como por ejemplo un archivo de texto donde si ya sabemos su peso podríamos consultar qué tamaño tiene antes de siquiera descargarlo o consultar de qué tipo de archivo estamos hablando. Estas y más información o mejor dicho meta-información está disponible para nosotros a partir del estado 2(dos) en donde ya tenemos los headers de la respuesta del servidor.

Headers

Las cabeceras (en inglés *headers*) HTTP permiten al cliente y al servidor enviar información adicional junto a una petición o respuesta. Una cabecera de petición está compuesta por su nombre (no sensible a las mayúsculas) seguido de dos puntos ':', y a continuación su valor (sin saltos de línea). Los espacios en blanco a la izquierda del valor son ignorados⁵



Request URL	Request Method	Status Code
http://localhost/drupal-7/user	GET	200 OK

Request Headers (10)

Response Headers (view source)

- Cache-Control: no-cache, must-revalidate, post-check=0, pre-check=0
- Connection: Keep-Alive
- Content-Language: en
- Content-Type: text/html; charset=utf-8
- Date: Thu, 17 Oct 2013 10:43:04 GMT
- ETag: "1382006584"
- Expires: Thu, 17 Oct 2013 10:53:04 +0000
- Keep-Alive: timeout=5, max=100
- Last-Modified: Thu, 17 Oct 2013 10:43:04 +0000
- Server: Apache/2.2.23 (Unix) mod_ssl/2.2.23 OpenSSL/0.9.8y DAV/2 PHP/5.4.10
- Transfer-Encoding: chunked
- X-Frame-Options: SAMEORIGIN
- X-Generator: Drupal 7 (http://drupal.org)
- X-Powered-By: PHP/5.4.10

Asumamos por un momento que alguno de los headers no nos gustan , no eran lo que estábamos esperando a cambio. En tal caso, estamos a tiempo de **abortar la solicitud** usando el método **abort()**. Hay que tener en cuenta que usando este método automáticamente reinicia el objeto XHR y perdemos toda configuración hecha hasta el momento.

Load

Este evento se dispara cuando la propiedad `readyState` es igual a 4. Hay que recordar nuevamente que esto no implica que el pedido haya sido exitoso por lo que aún tenemos que realizar una mínima comprobación para saber el status del mismo :

```
var xhr = new XMLHttpRequest()
xhr.open("GET", "miArchivo.ext")
xhr.addEventListener("load", function(){
    if (xhr.status == 200) {
        //...
    }
})
```

Este es un buen momento para revisar la respuesta que obtuvimos ya que pasamos por todos los estados, entonces deberíamos tener el contenido del recurso que fuimos a pedir.

Response

La propiedad `response` de un objeto XHR se va a completar cuando el pedido haya sido despachado y una vez que se haya descargado la información necesaria. La misma puede estar codificada en varios formatos : JSON, Document, Blob, ArrayBuffer y DOMString dependiendo cómo configuremos la propiedad `responseType`.

Send

El método `send` es el que nos permite enviar el pedido una vez que ya lo hayamos configurado. El mismo nos permite enviar parámetros en su interior como argumentos siempre y cuando la solicitud se haya realizado por el método POST :

```
var xhr = new XMLHttpRequest()
xhr.addEventListener("readystatechange", ()=>{
    console.log(xhr.readyState)
```



```
    })  
    xhr.addEventListener("load",function(){  
        if (xhr.status == 200) {  
            console.log(xhr.response)  
        }  
    })  
    xhr.open("GET","url.com")  
    xhr.send()
```

1. <https://developer.mozilla.org/es/docs/Web/Guide/AJAX>
2. Tengamos en cuenta que hay muchos namespaces para la misma API dentro de IE con lo cual el nombre del ActiveX Object puede variar. Ref.:
<https://blogs.msdn.microsoft.com/xmlteam/2006/10/23/using-the-right-version-of-msxml-in-internet-explorer/>
3. https://developer.mozilla.org/es/docs/Web/Guide/AJAX/Primeros_Pasos
4. <http://lemoncode.net/lemoncode-blog/2018/1/29/javascript-asincrono#el-modelo-de-javascript>
5. <https://developer.mozilla.org/es/docs/Web/HTTP/Headers>