

# JAVASCRIPT FRONTEND AVANZADO

## CLASE 7 : PROMISES

### PROMESAS

El objeto **Promise** (Promesa) es usado para computaciones asíncronas. Una promesa representa un valor que puede estar disponible ahora, en el futuro, o nunca . Una Promise (promesa en castellano) entonces, es un objeto que representa la terminación o el fracaso eventual de una operación asíncrona. Por qué deberíamos usarlas?

Pongamos un ejemplo sencillo de pedidos XHR dependientes entre sí. Imaginemos que queremos pedir los datos de los usuarios que están ahora mismo en nuestra aplicación :

```
var xhr = new XMLHttpRequest()
xhr.open("get","https://jsonplaceholder.typicode.com/users")
xhr.addEventListener("load",function(e){
  if (xhr.status == 200) {
    //Parseamos el string que nos devuelve la API primero
    var usuarios = JSON.parse(xhr.response)
    console.log(usuarios)
  }
})
xhr.send()
```

Una vez que obtuvimos los datos de los usuarios, podemos sacarles su ID . Con esta información ahora podemos entonces pedir todos los Posts que hizo ese usuario dentro de nuestra aplicación :

```
var xhr = new XMLHttpRequest()
xhr.open("get","https://jsonplaceholder.typicode.com/users")
xhr.addEventListener("load",function(){
  if (xhr.status == 200) {
    var usuarios = JSON.parse(xhr.response) //Parseamos el string que nos
    devuelve la API primero
    var xhr_posts = new XMLHttpRequest()
```

```
xhr_posts.open("get","https://jsonplaceholder.typicode.com/posts?userId="+usuarios[0].id)
    xhr_posts.addEventListener("load",function(){
        if (xhr_posts.status == 200) {
            var posts = JSON.parse(xhr_posts.response)
            //Obtenemos un listado de los posts de ese usuario solo
            console.log(posts)
        }
    })
    xhr_posts.send()
}
})
xhr.send()
```

Y ahora que tenemos la información de cada Post, podemos ir a consultar a la API si algún usuario le hizo un comentario:

```
var xhr = new XMLHttpRequest()
xhr.open("get","https://jsonplaceholder.typicode.com/users")
xhr.addEventListener("load",function(){
    if (xhr.status == 200) {
        var usuarios = JSON.parse(xhr.response) //Parseamos el string que nos devuelve la API primero
        var xhr_posts = new XMLHttpRequest()
        xhr_posts.open("get","https://jsonplaceholder.typicode.com/posts?userId="+usuarios[0].id)
        xhr_posts.addEventListener("load",function(){
            if (xhr_posts.status == 200) {
                var posts = JSON.parse(xhr_posts.response)
                var xhr_comment = new XMLHttpRequest()
                xhr_comment.open("get",
                    "https://jsonplaceholder.typicode.com/comments?postId="+posts[0].id)
                xhr_comment.addEventListener("load",function(){
                    if (xhr_comment.status == 200) {
                        /*Pyramid of DOOM*/ var comments = JSON.parse(xhr_comment.response)
                        console.log(comments)
                    }
                })
                xhr_comment.send()
            }
        })
        xhr_posts.send()
    }
})
xhr.send()
```

Podemos observar cómo se genera una dependencia de código asíncrono el cual tiene forma de triángulo, ya que un pedido depende de la respuesta del anterior para poder ejecutarse; a esto se lo conoce como Pyramid of DOOM ó Callback of Hell.

Cómo podemos evitar este tipo de patrón? Con Promesas.

Las Promesas nos permiten escribir código asíncrono de una manera encadenable y de forma tal que el mismo se lea como si fuera secuencial, es decir líneas después de líneas y no callbacks dentro de callbacks. Esencialmente, una promesa es un objeto devuelto al cual enganchas las funciones callback, en vez de pasar funciones callback a una función.

## GARANTIAS

A diferencia de las funciones callback pasadas al viejo estilo, una promesa viene con algunas garantías:

- Las funciones callback nunca serán llamadas antes de la terminación de la ejecución actual del bucle de eventos de JavaScript.
- Las funciones callback añadidas con `.then` serán llamadas *después* del éxito o fracaso de la operación asíncrona, como arriba.
- Pueden ser añadidas múltiples funciones callback llamando a `.then` varias veces, para ser ejecutadas independientemente en el orden de inserción.

Pero el beneficio más inmediato de las promesas es el encadenamiento <sup>2</sup>.

## ENCADENAMIENTO

Una necesidad común es el ejecutar dos o más operaciones asíncronas seguidas, donde cada operación posterior se inicia cuando la operación previa tiene éxito, con el resultado del paso previo. Logramos esto creando una cadena de promesas.

## SINTAXIS

```
new Promise( /* ejecutor */ function(resolver, rechazar) { ... } );
```

**Ejecutor** : Una función con los argumentos `resolver` y `rechazar`. La función ejecutor es ejecutada inmediatamente por la implementación de la Promesa, pasándole las funciones `resolver` y `rechazar` (el ejecutor es llamado incluso antes de que el constructor de la Promesa devuelva el objeto creado). Las funciones `resolver` y `rechazar`, al ser llamadas, resuelven o rechazan la promesa, respectivamente. Normalmente el ejecutor inicia un trabajo asíncrono, y luego, una vez que es completado, llama a la función `resolver` para resolver la promesa o la rechaza si ha ocurrido un error.

Si un error es lanzado en la función ejecutor, la promesa es rechazada y el valor de retorno del ejecutor es rechazado.

Una Promesa se encuentra en uno de los siguientes estados:

- **pendiente (pending)**: estado inicial, no cumplida o rechazada.
- **cumplida (fulfilled)**: significa que la operación se completó satisfactoriamente.
- **rechazada (rejected)**: significa que la operación falló <sup>1</sup>.

Una promesa pendiente puede ser *cumplida* con un valor, o *rechazada* con una razón (error).

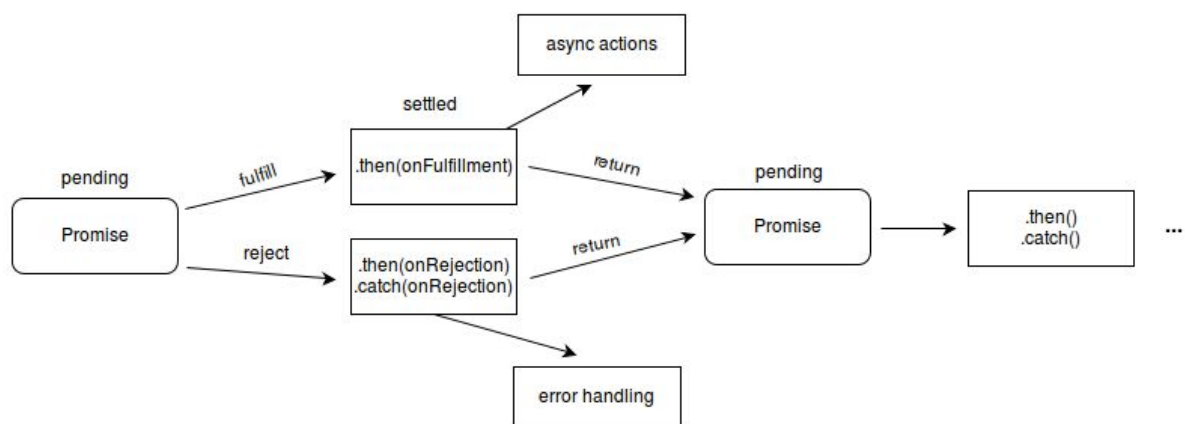
Cuando cualquiera de estas dos opciones sucede, los métodos asociados, encolados por el método *then* de la promesa, son llamados. (Si la promesa ya ha sido cumplida o rechazada en el momento que es anexado su correspondiente manejador, el manejador será llamado, de tal manera que no exista una condición de carrera entre la operación asíncrona siendo completada y los manejadores siendo anexados).

```
let miPrimeraPromise = new Promise((resolve, reject) => {
  // Llamamos a resolve(...) cuando lo que estábamos
  // haciendo finaliza con éxito, y reject(...) cuando falla.
  // En este ejemplo, usamos setTimeout(...) para simular
  // código asíncrono.
  // En la vida real, probablemente uses algo como XHR o
  // una API HTML5.
  setTimeout(function(){
    resolve("¡Éxito!"); // ¡Todo salió bien!
  }, 250);
});
```

Cuando una promesa se completa o rechaza, la misma disparará el listener registrado en su método *then* (o alternativamente en un *catch* si la misma tuvo algún error y no está configurado el segundo parámetro del *then*).

```
miPrimeraPromise
  .then(function(val){
    console.log(val)
  })
  .catch(function(err) {
    console.error(err)
  })
```

Como los métodos [Promise.prototype.then\(\)](#) y [Promise.prototype.catch\(\)](#) retornan promesas, éstas pueden ser encadenadas.



## FETCH API

La API de Fetch proporciona una interfaz para recuperar recursos (incluso a través de la red). Le parecerá familiar a cualquiera que haya usado XMLHttpRequest, pero la nueva API proporciona un conjunto de características más potente y flexible.

Fetch proporciona una definición genérica de los objetos de Request y Response (y otras cosas relacionadas con las solicitudes de red). Esto les permitirá ser utilizados donde sea que se necesiten en el futuro, ya sea para service workers, API de caché y otras cosas similares que manejan o modifican solicitudes y respuestas, o cualquier tipo de caso de uso que pueda requerir que generes tus propias respuestas programáticamente.

También proporciona una definición de conceptos relacionados, como CORS y la semántica del encabezado de origen HTTP, suplantando sus definiciones por separado en otros lugares.

Para realizar una solicitud y obtener un recurso, use el método `GlobalFetch.fetch`. Se implementa en múltiples interfaces, específicamente `Window` y `WorkerGlobalScope`. Esto lo hace disponible en prácticamente cualquier contexto en el que desee buscar recursos.

El método `fetch()` toma un argumento obligatorio, la ruta al recurso que desea recuperar. Devuelve una Promesa que resuelve en un objeto `Response` a esa solicitud, ya sea que sea exitosa o no. También puede opcionalmente pasar un objeto de opciones `init` como segundo argumento.

Una vez que se recupera una respuesta, hay varios métodos disponibles para definir el contenido del cuerpo y cómo se debe manejar.

Puede crear una solicitud y respuesta directamente utilizando los constructores `Request()` y `Response()`, pero es poco probable que lo haga directamente. En cambio, es más probable que se creen como resultados de otras acciones de la API (por ejemplo, `FetchEvent.respondWith` de los service workers) <sup>3</sup>.

## REQUEST

La interfaz de `Request` de la API de Fetch representa una solicitud de recursos.

Puede crear un nuevo objeto `Request` utilizando el constructor `Request.Request()`, pero es más probable que encuentre un objeto `Request` que se devuelve como resultado de otra operación de API, como un service worker `FetchEvent.request` <sup>4</sup>.

Esta interfaz cuenta con varias propiedades configurables para realizar un pedido saliente :

PROPIEDAD	DESCRIPCIÓN
<b>method</b>	Contiene el método de la solicitud
<b>url</b>	Contiene la URL de la solicitud
<b>headers</b>	Contiene un objeto Headers asociado
<b>body</b>	Contiene un objeto Body asociado

## RESPONSE

La interfaz de `Response` de la API de Fetch representa la respuesta a una solicitud.

Puede crear un nuevo objeto `Response` utilizando el constructor `Response.Response()`, pero es más probable que encuentre un objeto `Response` que se devuelve como resultado de otra operación de la API, por ejemplo, un service worker `FetchEvent.respondWith` o un simple `GlobalFetch`. ha podido recuperar() <sup>5</sup>.

Esta interfaz cuenta con varias propiedades de lectura y métodos para la transformación del cuerpo de la respuesta en cuestión :

PROPIEDAD	DESCRIPCIÓN
<b>headers</b>	Contiene un objeto Headers asociado

<b>ok</b>	Contiene un booleano correspondiente al estado final de la solicitud
<b>status</b>	Contiene el código HTTP asociado a la respuesta
<b>json()</b>	Transforma el Body de la respuesta en formato JSON y retorna una promesa
<b>blob()</b>	Transforma el Body de la respuesta en formato Blob y retorna una promesa
<b>arrayBuffer()</b>	Transforma el Body de la respuesta en formato ArrayBuffer y retorna una promesa

## GEOLOCATION API

La interfaz de geolocalización representa un objeto capaz de obtener programáticamente la posición del dispositivo. Le da acceso al contenido web a la ubicación del dispositivo <sup>6</sup>.

Esto permite que un sitio web o una aplicación ofrezca resultados personalizados según la ubicación del usuario. Un objeto con esta interfaz se obtiene utilizando la propiedad `navigator.geolocation` implementada por el objeto Navigator.

### Contexto seguro

Esta función sólo está disponible en contextos seguros (HTTPS) en algunos o en todos los navegadores compatibles.

### Nota

Por razones de seguridad, cuando una página web intenta acceder a información de ubicación, se le notifica al usuario y se le solicita que conceda permiso. Tenga en cuenta que cada navegador tiene sus propias políticas y métodos para solicitar este permiso.

Para obtener la ubicación actual del usuario, puede llamar al método **`getCurrentPosition()`**. Esto inicia una solicitud asíncrona para detectar la posición del usuario y consulta el hardware de posicionamiento para obtener información actualizada. Cuando se determina la posición, se ejecuta la función callback definida.

Opcionalmente, puede proporcionar una segunda función callback para que se ejecute si ocurre un error. Un tercer parámetro opcional es un objeto de opciones donde puede establecer la edad máxima de la posición devuelta, el tiempo para esperar una solicitud y si desea una alta precisión para la posición.

```
navigator.geolocation.getCurrentPosition(function(position) {  
    console.log(position.coords.latitude, position.coords.longitude);  
});
```

#### Nota

Por defecto, `getCurrentPosition ()` intenta responder lo más rápido posible con un resultado de baja precisión. Es útil si necesita una respuesta rápida independientemente de la precisión. Los dispositivos con un GPS, por ejemplo, pueden tardar un minuto o más para obtener una corrección de GPS, por lo que pueden devolverse datos menos precisos (ubicación de IP o wifi) a `getCurrentPosition ()`.

Si los datos de posición cambian (ya sea por movimiento del dispositivo o si llega información geográfica más precisa), puede configurar una función callback que se llame con esa información de posición actualizada.

Esto se hace usando la función **`watchPosition ()`**, que tiene los mismos parámetros de entrada que `getCurrentPosition ()`. La función callback se llama varias veces, lo que permite al navegador actualizar su ubicación a medida que se mueve o proporciona una ubicación más precisa a medida que se utilizan diferentes técnicas para geolocalizarlo. La función callback de error, que es opcional tal como lo es para `getCurrentPosition ()`, se puede invocar repetidamente <sup>7</sup>.

1. [https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos\\_globales/Promise](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Promise)
2. [https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Usar\\_promesas](https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Usar_promesas)
3. [https://developer.mozilla.org/en-US/docs/Web/API/Fetch\\_API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API)
4. <https://developer.mozilla.org/en-US/docs/Web/API/Request>
5. <https://developer.mozilla.org/en-US/docs/Web/API/Response>
6. <https://developer.mozilla.org/en-US/docs/Web/API/Geolocation>
7. [https://developer.mozilla.org/en-US/docs/Web/API/Geolocation/Using\\_geolocation](https://developer.mozilla.org/en-US/docs/Web/API/Geolocation/Using_geolocation)