UD 07. UTILIZACIÓN DE MECANISMOS DE COMUNICACIÓN ASINCRONA.

Índice de contenido

1. Asincronismo 2
1.1 Event loop. 3
1.2 Cola de tareas (Task Queue) 3
1.3 Callback. 3
1.4 Promesas. 3
1.5 Async / Await. 4
2. Métodos de petición HTTP 4
3. ¿Qué es AJAX? 5
4. Tecnologías presentes en AJAX 5
5. Funcionamiento de aplicación Web clásica VS aplicación Web AJAX 6
5.1 Aplicación Web clásica 6
5.2 Aplicación Web AJAX 6
6. Objeto XMLHttpRequest 6
7. Principales métodos 7
7.1 Atributos 7
7.2 Métodos 7
7.3 Eventos 8
8. Forma más común de utilizar XMLHttpRequest 8
8.1 Instanciando el objeto 8
8.2 Comportamiento evento onreadystatechange 9
9. FETCH 10
10. Material adicional 12
11. Bibliografía 12

UTILIZACIÓN DE MECANISMOS DE COMUNICACIÓN ASINCRONA

1. ASINCRONISMO.

La principal diferencia entre programación síncrona y asíncrona es que la síncrona ejecuta las tareas una después de la otra, esperando a que cada una termine antes de iniciar la siguiente. En cambio, la asíncrona permite que una tarea se ejecute en segundo plano sin bloquear el programa principal, de modo que puede continuar con otras operaciones mientras la tarea asíncrona se completa. Esta es otra diferencia entre java y JavaScript.

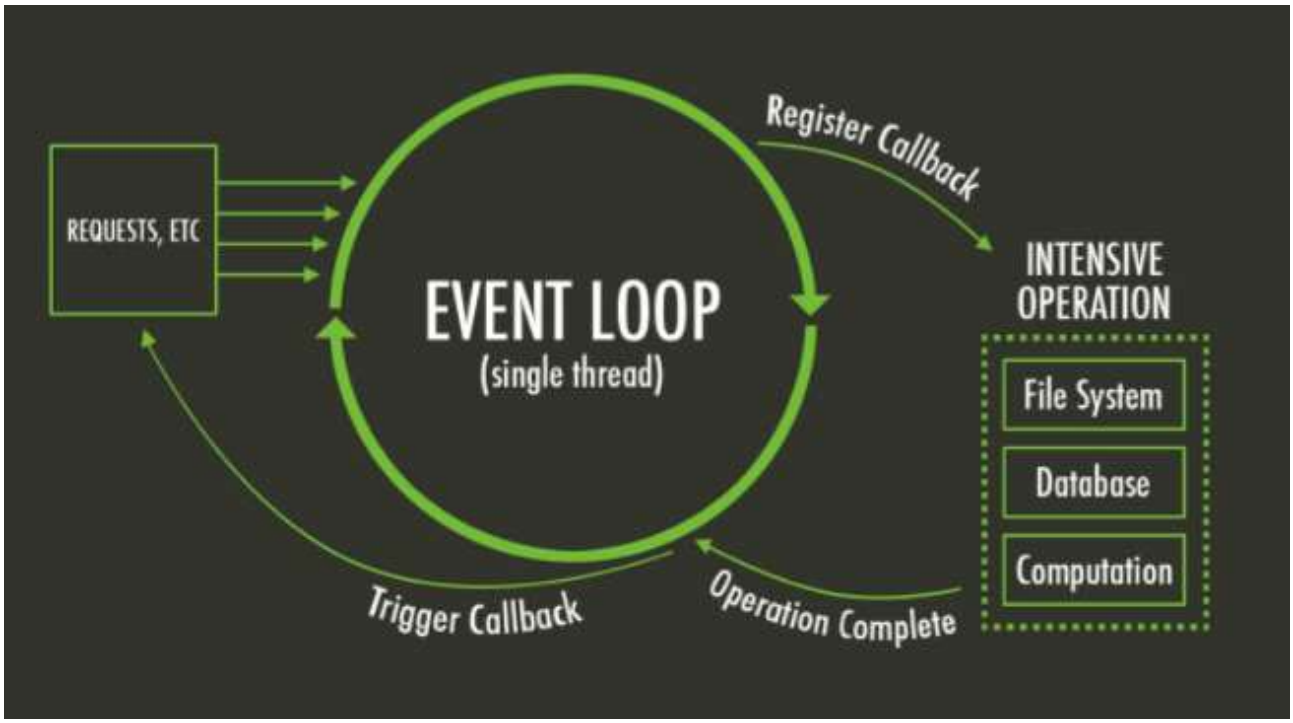
La asincronía es uno de los pilares fundamentales de Javascript, ya que es un lenguaje de programación de un sólo subproceso o hilo (single thread), lo que significa que sólo puede ejecutar una cosa a la vez.

JavaScript es asíncrono gracias al event loop y la cola de tareas. Esto permite que operaciones como peticiones HTTP o temporizadores se ejecuten sin bloquear el hilo principal. Estas operaciones asíncronas son gestionadas por el navegador o el entorno de Node.js y, cuando terminan, sus callbacks se colocan en la cola para ser ejecutados cuando el hilo principal esté libre.

Imagina que solicitas datos de una API. Dependiendo de la situación, el servidor puede tardar un tiempo en procesar la solicitud mientras bloquea el hilo principal y hace que la página web no responda.

Javascript usa un modelo asíncrono y no bloqueante, con un loop de eventos implementado en un sólo hilo, (single thread) para operaciones de entrada y salida (input/output), aunque posee funciones que son bloqueantes como el comando alert.

Gracias a esta solución, Javascript es altamente concurrente a pesar de emplear un sólo hilo.



Para controlar la asincronía, JavaScript cuenta con algunos mecanismos:

- Callbacks.
- Promises.
- Async / Await.

1.1 Event loop.

A pesar de que JavaScript es un lenguaje monohilo (single-threaded), en JavaScript existe el Event Loop (bucle de eventos), que es un mecanismo que permite la ejecución de código asíncrono y no bloqueante.

1.2 Cola de tareas (Task Queue)

JavaScript gestiona las operaciones asíncronas utilizando varias colas de tareas, cada una con una prioridad diferente. Estas colas incluyen la cola de macrotareas y la cola de microtareas.

Las macrotareas y microtareas son dos tipos de colas de ejecución en JavaScript que gestionan las tareas asíncronas a través del bucle de eventos. Las macrotareas son tareas de mayor prioridad como los eventos de temporizador (\(setTimeout\), \(setInterval\)) y los eventos de interfaz de usuario (clics), mientras que las microtareas son de menor prioridad, pero se ejecutan inmediatamente después de completar la tarea actual y la pila de llamadas se vacía, como en las promesas (\(.then\), \(async/await\)).

- Macrotareas: Son las tareas que se colocan en la cola de tareas principal. Incluyen:
  - setTimeout
  - setInterval
  - Eventos del DOM (como clicks)
  - Peticiones HTTP (fetch, XMLHttpRequest)
- Microtareas: Se colocan en la cola de microtareas y tienen prioridad sobre las macrotareas. Esto significa que después de cada macrotarea, el motor de JavaScript procesará todas las microtareas antes de continuar con la siguiente macrotarea. Incluyen:
  - Promesas (.then, .catch, .finally)

1.3 Callback.

Una callback (llamada de vuelta) es una función que se ejecutará después de que otra lo haga.

Es un mecanismo para asegurar que cierto código no se ejecute hasta que otro haya terminado.

Al ser JavaScript un lenguaje orientado a eventos, las callbacks son una buena técnica para controlar la asíncronía, sin embargo, aparece el callback hell, callback del infierno, o pirámide del infierno.

El "callback hell" (infierno de callbacks) es un antipatrón de programación que surge al anidar múltiples funciones de callback, lo que crea un código profundo y en forma de pirámide difícil de leer, mantener y depurar

1.4 Promesas.

Una promesa es un objeto que representa el resultado de una operación asíncrona y tiene 3

estados posibles:

- Pendiente.
- Resuelta.
- Rechazada.

Tienen la particularidad de que se pueden encadenar (then), siendo el resultado de una promesa, los datos de entrada de otra posible función.

Las promesas mantienen un código más legible y mantenible que las callbacks, además tienen un mecanismo para la detección de errores (catch) que es posible usar en cualquier parte del flujo de datos.

1.5 Async / Await

Las promesas fueron una gran mejora respecto a las callbacks para controlar la asincronía en JavaScript, sin embargo pueden llegar a ser muy verbosas a medida que se requieran más y más métodos .then().

Las funciones asíncronas (async / await) surgen para simplificar el manejo de las promesas.

La palabra async declara una función como asíncrona e indica que una promesa será automáticamente devuelta.

Podemos declarar como async funciones con nombre, o anónimas.

La palabra await debe ser usado siempre dentro de una función declarada como async y esperará de forma asíncrona y no bloqueante a que una promesa se resuelva o rechace.

2. MÉTODOS DE PETICIÓN HTTP

HTTP define un conjunto de métodos de petición para indicar la acción que se desea realizar para un recurso determinado. Aunque estos también pueden ser sustantivos, estos métodos de solicitud a veces son llamados HTTP verbs. Cada uno de ellos implementan una semántica diferente, pero algunas características similares son compartidas por un grupo de ellos:

Los principales son:

- GET : El método GET solicita una representación de un recurso específico. Las peticiones que usan el método GET sólo deben recuperar datos.
- POST : El método POST se utiliza para enviar una entidad a un recurso en específico, causando a menudo un cambio en el estado o efectos secundarios en el servidor.
- PUT : El modo PUT reemplaza todas las representaciones actuales del recurso de destino con la carga útil de la petición.
- DELETE : El método DELETE borra un recurso en específico.

- PATCH : El método PATCH es utilizado para aplicar modificaciones parciales a un recurso.

3. ¿QUÉ ES AJAX?

En informática, AJAX es el acrónimo de “Asynchronous Javascript And XML” (JavaScript asíncrono y XML).

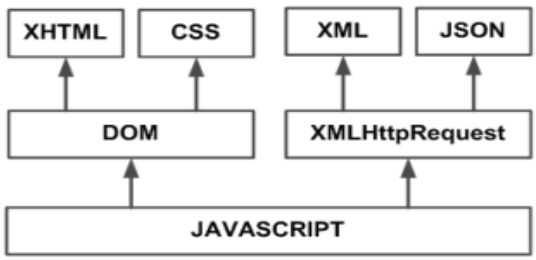
AJAX en si no es una tecnología, sino un conjunto de tecnologías.

Tenéis más información en https://es.wikipedia.org/wiki/AJAX

4. TECNOLOGÍAS PRESENTES EN AJAX

Las tecnologías presentes en AJAX son:

- XHTML y CSS para la presentación de la página.
- DOM para la manipulación dinámica de elementos de la página.
- Formatos de intercambio de información como JSON o XML.
- El objeto XMLHttpRequest, para el intercambio asíncrono de información (es decir, sin recargar la página).
- Javascript, para aplicar las anteriores tecnologías.



La forma de trabajar es la siguiente: Javascript se encarga de unir todas las tecnologías. Para manipular la parte de representación de la página utiliza DOM (así manipula el XHTML y el CSS).

Para realizar peticiones asíncronas usa el objeto XMLHttpRequest. Este objeto intercambia información que son simplemente cadenas de texto. Cuando se quieren formatear objetos más complejos, se suele utilizar JSON o XML.

Durante el curso utilizaremos habitualmente JSON para intercambiar la información.

La notación JSON para los arrays asociativos se compone de tres partes:

- Los contenidos del array asociativo se encierran entre llaves ({ y })
- Los elementos del array se separan mediante una coma (,)
- La clave y el valor de cada elemento se separan mediante dos puntos (:)
- Si la clave no contiene espacios en blanco, es posible prescindir de las comillas que encierran sus contenidos.
- Sin embargo, las comillas son obligatorias cuando las claves pueden contener espacios en blanco.

5. FUNCIONAMIENTO DE APLICACIÓN WEB CLÁSICA VS APLICACIÓN WEB AJAX

5.1 Aplicación Web clásica

En una aplicación Web clásica:

1) El cliente hace una petición al servidor.
2) El servidor recibe la petición.
3) El servidor procesa la petición y genera una nueva página con la petición procesada. (Ejemplo, se añade un post a un foro).
4) El cliente recibe la nueva página completa y la muestra.

5.2 Aplicación Web AJAX

En una aplicación Web AJAX:

1) El cliente hace una petición asíncrona al servidor.
2) El servidor recibe la petición.
3) El servidor procesa la petición y responde asíncronamente al cliente.
4) El cliente recibe la respuesta y con ella modifica dinámicamente los elementos afectados de la página sin recargarla completamente.

Las aplicaciones Web AJAX son mejores ya que reducen la cantidad de información a intercambiar (no se envía la página entera, sino que se modifica solo lo que interesa) y a su vez al usuario final le da una imagen de mayor dinamismo, viendo una página web como una aplicación de escritorio.

Como único contra, el diseño de aplicaciones Web AJAX es ligeramente más complicado que el desarrollo de aplicaciones Web clásicas.

6. OBJETO XMLHTTPREQUEST

En esta unidad didáctica hablaremos de cómo funciona el objeto XMLHttpRequest, imprescindible para utilizar AJAX. Este objeto nos permitirá hacer peticiones asíncronas y se encargará de avisarnos cuando se reciba su respuesta.

7. PRINCIPALES MÉTODOS

En https://es.wikipedia.org/wiki/XMLHttpRequest se puede observar una definición del objeto y sus principales atributos y métodos.

Tomado de Wikipedia:

7.1 Atributos

Atributo	Descripción
readyState	Devuelve el estado del objeto como sigue: 0 = sin inicializar, 1 = abierto, 2 = cabeceras recibidas, 3 = cargando y 4 = completado.
responseBody	*(Level 2)* Devuelve la respuesta como un array de bytes
responseText	Devuelve la respuesta como una cadena
responseXML	Devuelve la respuesta como XML. Esta propiedad devuelve un objeto documento XML, que puede ser examinado usando las propiedades y métodos del árbol del Document Object Model.
status	Devuelve el estado como un número (p. ej. 404 para "Not Found" y 200 para "OK").
statusText	Devuelve el estado como una cadena (p. ej. "Not Found" o "OK").

7.2 Métodos

Método	Descripción
abort()	Cancela la petición en curso
getAllResponseHeaders()	Devuelve el conjunto de cabeceras HTTP como una cadena.
getResponseHeader(nombreCabecera)	Devuelve el valor de la cabecera HTTP especificada.
open(método, URL, [asíncrono, [nombreUsuario, [clave]]])	Especifica el método, URL y otros atributos opcionales de una petición. El parámetro de método puede tomar los valores "GET", "POST", o "PUT" ("GET" y "POST" son dos formas para solicitar datos, con "GET" los parámetros de la petición se codifican en la URL y con "POST" en las cabeceras de HTTP). El parámetro *URL* puede ser una URL relativa o completa.

	El parámetro *asíncrono* especifica si la petición será gestionada asíncronamente o no. Un valor *true* indica que el proceso del script continúa después del método send(), sin esperar a la respuesta, y *false* indica que el script se detiene hasta que se complete la operación, tras lo cual se reanuda la ejecución. En el caso asíncrono se especifican manejadores de eventos, que se ejecutan ante cada cambio de estado y permiten tratar los resultados de la consulta una vez que se reciben, o bien gestionar eventuales errores.
send([datos])	Envía la petición
setRequestHeader(etiqueta, valor)	Añade un par etiqueta/valor a la cabecera HTTP a enviar.

7.3 Eventos

Propiedad	Descripción
onreadystatechange	Evento que se dispara con cada cambio de estado.
onabort	*(Level 2)* Evento que se dispara al abortar la operación.
onload	*(Level 2)* Evento que se dispara al completar la carga.
onloadstart	*(Level 2)* Evento que se dispara al iniciar la carga.
onprogress	*(Level 2)* Evento que se dispara periódicamente con información de estado.

8. FORMA MÁS COMÚN DE UTILIZAR XMLHTTPREQUEST

8.1 Instanciando el objeto

En primer lugar, indicar que debemos inicializar el objeto.

```
httpRequest = new XMLHttpRequest();
```

Esto es válido para la mayoría de navegadores actuales.

Si queremos compatibilidad con navegadores antiguos que soporte ActiveX (Tipo Internet Explorer 6) se puede hacer una función más compleja para obtener el objeto.

```
function obtainXMLHttpRequest()
{
      var httpRequest;
```

```
	if (window.XMLHttpRequest){
		//El explorador implementa la interfaz de forma nativa
		httpRequest = new XMLHttpRequest();
	}
	else if (window.ActiveXObject){
		//El explorador permite crear objetos ActiveX
		try {
			httpRequest = new ActiveXObject("MSXML2.XMLHTTP");
		} catch (e) {
			try {
				httpRequest = new ActiveXObject("Microsoft.XMLHTTP");
			} catch (e) {}
		}
	}
	// Si no se puede crear, devolvemos false. En caso contrario, devolvemos
el objeto
	if (!httpRequest){
		return false;
	}
	else{
		return httpRequest;
	}
}
```

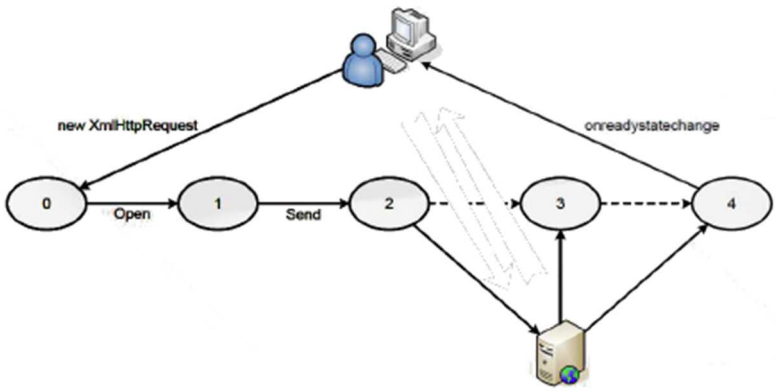
8.2 Comportamiento evento onreadystatechange

Tras ello, decidiremos el comportamiento del evento “onreadystatechange”, evento que se producirá cada vez que haya producido un cambio en el atributo “ready”.

Antes de hacer nada, deberemos explicar 3 métodos del objeto:

- open(Metodo,URL,[tipo],[usuario],[passwd]). Este método recibe que método de comunicación utiliza en la petición (GET o POST) y que ULR se aplica.
- send([datos]). Este método ha de hacerse posteriormente a un open, nunca antes. Envía la información a la URL especificada en open.
- setRequestHeader que indicara el formato de las cabeceras enviadas.

También debemos explicar el atributo readyState: este atributo puede poseer los siguientes valores:



- 0 al inicializarse el objeto.
- 1 al abrirse una conexión (al usar el método open).
- 2 al hacer una petición (uso de send).
- 3 mientras se está recibiendo información de la petición.
- 4 cuando la petición se ha completado.

¿Cómo lo aplicamos en código? El evento onreadystatechange cada vez que se produzca comprobará en qué estado nos encontramos y hará lo planeado para ese estado. Generalmente el estado más utilizado es el 4, donde se ha completado la operación.

Ejemplo: Un ejemplo de todo esto donde se hace una petición y al finalizarse si el estado es correcto (200) se actualiza un div con id=capa en el código.

```
// Abrimos la conexión
httpRequest.open("POST","ajax.php",true);
// Indicamos como seran las cabeceras
httpRequest.setRequestHeader("Content-Type","application/x-www-form-
urlencoded");
// Definimos el comportamiento de onreadystatechange
httpRequest.onreadystatechange=function(){
      if (httpRequest.readyState==1) {
            document.getElementById('capa').innerHTML="CARGANDO...";
      }
      // Si se ha completado
      if (httpRequest.readyState==4) {
            // Si es correcto el status
            if (httpRequest.status==200){
            // de httpRequest.responseText obtenemos la cadena con la respuesta
                  document.getElementById('capa').innerHTML=httpRequest
.responseText;
            }
      }
}
// Enviamos la acción
httpRequest.send("accion="+accion);
```

En este ejemplo, abrimos con open, definimos el comportamiento del evento "onreadystatechange" y cuando está todo listo, enviamos la petición con send.

9. FETCH

La API Fetch proporciona una interfaz JavaScript para acceder y manipular partes del canal HTTP, tales como peticiones y respuestas. También provee un método global fetch() que proporciona una forma fácil y lógica de obtener recursos de forma asíncrona por la red.

Este tipo de funcionalidad se conseguía previamente haciendo uso de XMLHttpRequest. Fetch proporciona una alternativa mejorada que puede ser empleada fácilmente por otras tecnologías como Service Workers. Fetch también aporta un único lugar lógico en el que definir otros conceptos relacionados con HTTP como CORS y extensiones para HTTP.

```
fetch('http://example.com/movies.json')
  .then(response => response.json())
  .then(data => console.log(data));
fetch('http://example.com/movies.json')
  .then(response => response.json())
  .then(data => console.log(data));
```

Aquí estamos recuperando un archivo JSON a través de red e imprimiendo en la consola. El uso de fetch() más simple toma un argumento (la ruta del recurso que quieres obtener) y devuelve un objeto Promise conteniendo la respuesta, un objeto Response.

Esto es, por supuesto, una respuesta HTTP no el archivo JSON. Para extraer el contenido en el cuerpo del JSON desde la respuesta, usamos el método json() (definido en el mixin de Body, el cual está implementado por los objetos Request y Response).

Los datos se pueden enviar de 2 maneras, en la url en el campo data, o en el atributo body

Mediante la url

```
fetch('http://example.com?data='+datostarea, { JSON.stringify({
  title: "Hello World",
  body: "My POST request",
  userId: 900,
 }),

        method: 'POST',
        })
        .then(response => response.json())
        .then(data => {
        console.log('Éxito:', data);
        })
        .catch((error) => {
        console.error('Error:', error);
        });
    }
```

```
fetch("https://jsonplaceholder.typicode.com/posts", {
 method: "POST",
 body: JSON.stringify({
  title: "Hello World",
  body: "My POST request",
  userId: 900,
 }),
 headers: {
  "Content-type": "application/json; charset=UTF-8",
 },
})
.then((response) => response.json())
```

```
.then((json) => console.log(json));
```

10. MATERIAL ADICIONAL

[1] Intro to AJAX (Udacity) https://www.udacity.com/course/intro-to-ajax--ud110

11. BIBLIOGRAFÍA

[1] Referencia Javascript
http://www.w3schools.com/jsref/
[2] XMLHttpRequest
https://es.wikipedia.org/wiki/XMLHttpRequest
[3] Introducción a AJAX de Libroweb
https://librosweb.es/libro/ajax/
[4] JavaScript Asincrono
https://jonmircha.com/javascript-asincrono