

Bienvenidos

Clase 02. Programación Asíncrona
Ajax, fetch, Axios

Aplicaciones Móviles Y Cloud Computing

Objetivos de la clase

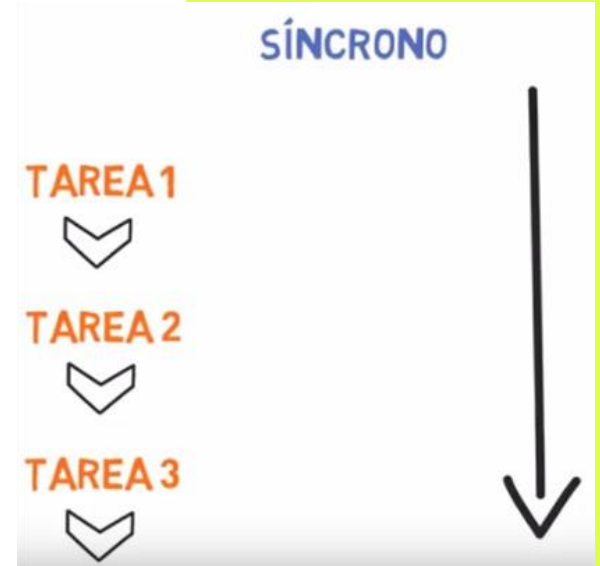
- Comprender el concepto de programación asíncrona
- Conocer algunas formas en que JavaScript maneja la asincronía
- Conocer Axios, Fetch, y Axios: herramientas de JavaScript para hacer requerimientos remotos



Sincronismo vs Asincronismo

Sincronismo

Cuando aprendiste a programar, entendiste que las instrucciones se ejecutaban **en cascada**. Es decir, que la tarea 1 debía finalizar para que pudiera comenzar la ejecución de la tarea 2, y la tarea 2 finalizar para ejecutar la tarea 3, etc.



Ejemplo de ejecución sincrónica

- ✓ En todo momento, sólo se están ejecutando las instrucciones de una sola de las funciones a la vez. O sea, debe finalizar una función para poder continuar con la otra.
- ✓ El fin de una función marca el inicio de la siguiente, y el fin de ésta, el inicio de la que le sigue, y así sucesivamente, describiendo una secuencia que ocurre en una única línea de tiempo.

```
function funA() {  
  console.log(1)  
  funB()  
  console.log(2)  
}  
function funB() {  
  console.log(3)  
  funC()  
  console.log(4)  
}  
function funC() {  
  console.log(5)  
}  
  
funA()  
  
//Al ejecutar la función funA()  
//se muestra lo siguiente por pantalla:  
1  
3  
5  
4  
2
```



Importante

Las operaciones síncronas son **bloqueantes**, esto significa que las otras tareas no pueden comenzar a ejecutarse hasta que la primera no haya terminado.

Asincronismo

Si lo que buscamos es que las tareas trabajen “en paralelo”, entonces debemos buscar la manera de programar instrucciones **asíncronas**, lo cual significa que cada una seguirá el hilo de resolución que considere su ritmo.

Hay que ser cautelosos al utilizarlas, ya que:

- ✓ No controlamos cuándo terminará, sólo cuándo comienza.
- ✓ Si una tarea depende del resultado de otra, habrá problemas, pues esperará su ejecución en paralelo

ASÍNCRONO

TAREA 1



TAREA 2



TAREA 3



Ejemplo de ejecución asincrónica

- ✓ En el ejemplo no se bloquea la ejecución normal del programa y se permite que este se siga ejecutando.
- ✓ La ejecución de la operación de escritura “comienza” e inmediatamente cede el control a la siguiente instrucción, que escribe por pantalla el mensaje de finalización.
- ✓ Cuando la operación de escritura termina, ejecuta el callback que informará por pantalla que la escritura se realizó con éxito.

```
const escribirArchivo = require('./escriArch.js')

console.log('inicio del programa')

// el creador de esta funcion la definió
// como no bloqueante. recibe un callback que
// se ejecutará al finalizar la escritura.
escribirArchivo('hola mundo', () => {
  console.log('terminé de escribir el archivo')
})

console.log('fin del programa')

// se mostrará por pantalla:
// > inicio del programa
// > fin del programa
// > terminé de escribir el archivo
```



Importante

Las operaciones asíncronas son **no bloqueantes**, esto significa que las tareas pueden irse ejecutando en paralelo y no esperar por las demás tareas. Así, la tarea número 3 podría terminar incluso antes que la tarea número 1

Callbacks

Un callback es una función como cualquier otra. La diferencia está en que ésta se pasa como parámetro (argumento) para poder ser utilizado por otra función.

Permite que entonces las funciones ejecuten operaciones adicionales dentro de sí mismas

Cuando pasamos un callback, lo hacemos porque no siempre sabemos qué queremos que se ejecute en cada caso de nuestra función.

Algunos ejemplos donde has utilizado callbacks son:

- ✓ El método `onClick` en frontend
- ✓ El método `forEach`
- ✓ El método **`map`** o `filter`



Convenciones en los Callbacks

```
const ejemploCallback = (error, resultado) => {  
  if (error) {  
    // hacer algo con el error!  
  } else {  
    // hacer algo con el resultado!  
  }  
};
```

Desde el lado del callback, debemos manejar eventualmente errores. Por este motivo, nos encontraremos muy a menudo con esta estructura.



Callbacks anidados

En algún momento el mundo laboral te exige hacer más que sólo una suma o una resta. Nos encontraremos con procesos que requieren operaciones de más pasos. Si nosotros trabajamos con callbacks, podemos encadenar un conjunto de operaciones secuenciales.

Así, un callback puede llamar a otro callback, y este puede llamar a otro callback, y este a otro...


```
1 // Callback Hell
2
3
4 a(function (resultsFromA) {
5   b(resultsFromA, function (resultsFromB) {
6     c(resultsFromB, function (resultsFromC) {
7       d(resultsFromC, function (resultsFromD) {
8         e(resultsFromD, function (resultsFromE) {
9           f(resultsFromE, function (resultsFromF) {
10             console.log(resultsFromF);
11           })
12         })
13       })
14     })
15   })
16 });
17
```

Callback Hell:

Cómo reconocerlos:

Si estás trabajando con callbacks y tu código comienza a tomar esta forma... ¡Mucho cuidado, hay que cambiar de estrategia!

```
pan.pourWater(function() {  
  range.bringToBoil(function() {  
    range.lowerHeat(function() {  
      pan.addRice(function() {  
        setTimeout(function() {  
          range.turnOff();  
          serve();  
        }, 15 * 60 * 1000);  
      });  
    });  
  });  
});
```



pyramid of doom

```
function register()  
{  
  if (!empty($_POST)) {  
    $msg = '';  
    if ($_POST['user_name']) {  
      if ($_POST['user_password_new']) {  
        if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {  
          if (strlen($_POST['user_password_new']) > 5) {  
            if (strlen($_POST['user_name']) < 64 && strlen($_POST['user_name']) > 1) {  
              if (preg_match('/^[a-zA-Z](\d{4})?/?$', $_POST['user_name']) {  
                $user = read_user($_POST['user_name']);  
                if (!isset($user['user_name'])) {  
                  if ($_POST['user_email']) {  
                    if (strlen($_POST['user_email']) < 65) {  
                      if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {  
                        create_user();  
                        $_SESSION['msg'] = 'You are now registered so please login';  
                        header('Location: ' . $_SERVER['PHP_SELF']);  
                        exit();  
                      } else $msg = 'You must provide a valid email address';  
                    } else $msg = 'Email must be less than 64 characters';  
                  } else $msg = 'Email cannot be empty';  
                } else $msg = 'Username already exists';  
              } else $msg = 'Username must be only a-z, A-Z, 0-9';  
            } else $msg = 'Username must be between 2 and 64 characters';  
          } else $msg = 'Password must be at least 6 characters';  
        } else $msg = 'Passwords do not match';  
      } else $msg = 'Empty Password';  
    } else $msg = 'Empty Username';  
    $_SESSION['msg'] = $msg;  
  }  
  return register_form();  
}
```





Break

¡Unos minutos y volvemos!



Universidad
Provincial del Sudoeste
Promoviendo el Desarrollo Armónico de la Región

Promesas

Es un objeto especial que nos permitirá **encapsular** una operación, la cual reaccionará a **dos posibles** situaciones dentro de una promesa:

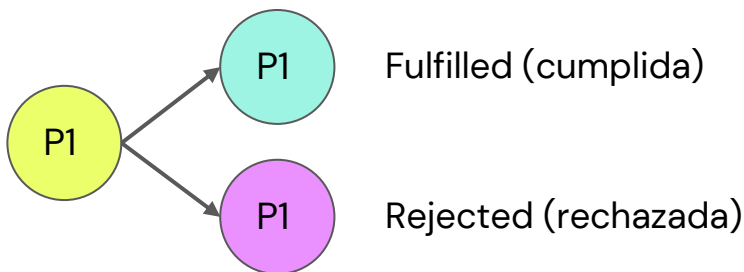
- ✓ ¿Qué debería hacer si la promesa se cumple?
- ✓ ¿Qué debería hacer si la promesa no se cumple?

The logo features the letters 'JS' in a bold, yellow, sans-serif font, enclosed within a black square. To the right of this square, the word 'Promises' is written in a large, black, sans-serif font. The entire logo is set against a solid yellow rectangular background.

Una promesa funciona muy similar al mundo real

Al prometerse algo, es una promesa en estado pendiente (pending). No sabemos cuándo se resolverá esa promesa. Sin embargo, cuando llega el momento, se nos notifica si la promesa se cumplió (**Fulfilled**, también lo encontramos como Resolved) o tal vez, a pesar del tiempo, al final nos notifiquen que la promesa no pudo cumplirse: se rechazó (**Rejected**).

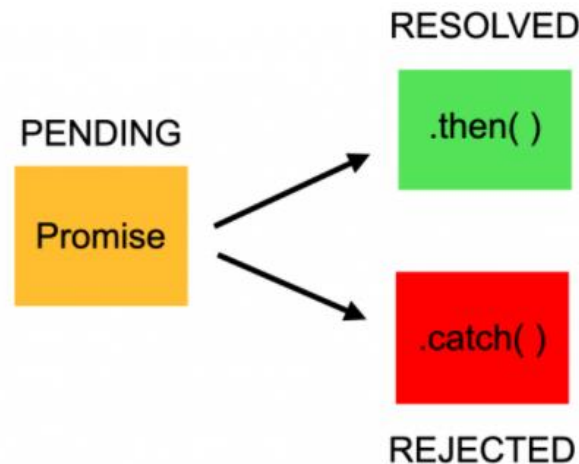
Promesas en Javascript



Utilizando promesas

Ahora que entendimos que hay dos formas de resolver una promesa (Resolved/Fulfilled o Rejected), debemos aprender cómo utilizar estos dos estados.

- ✓ Ejecutaremos la función que acabamos de crear para que se ejecute la promesa.
- ✓ Utilizaremos el operador **.then** para recibir el caso en el que la promesa **Sí se haya cumplido**
- ✓ Utilizaremos el operador **.catch** para recibir el caso en el que la promesa **No se haya cumplido**.



Async / Await



Problemas con `.then` y `.catch`

Cuando necesitamos más que sólo una operación para poder ejecutar algo asíncrono, no basta con el uso de una promesa solamente, sino que necesitamos un entorno completo para poder ejecutar dichas operaciones. El **`.then`** en este caso sólo nos sirve para encadenar las promesas y obtener sus resultados, pero no nos permite un entorno completo asíncrono para trabajar, por lo cual nos obliga a trabajar TODO dentro de ese scope

Además, el principal problema de los `.then` y `.catch` son su encapsulamiento excesivo, impidiendo o limitando que podamos acceder a los recursos de algunos resultados, variables, etc.

Problema



Universidad
Provincial del Sudoeste
Promoviendo el Desarrollo Armónico de la Región

Async / Await

Surge entonces en Javascript el soporte para Async – Await, unas palabras reservadas que, trabajando juntas, permiten gestionar un entorno asíncrono, resolviendo las limitantes del .then y .catch

- ✓ **async** se colocará al inicio de una función, indicando que **todo el cuerpo de esa función deberá ejecutarse de manera asíncrona**
- ✓ **await** servirá (como indica su nombre) para **esperar** por el resultado de la promesa y extraer su resultado.
- ✓ Al ser operaciones que podrían salir bien, **PERO TAMBIÉN MAL**, es importante encerrar el cuerpo en un bloque try {} catch {}

Solución



Universidad
Provincial del Sudoeste
Promoviendo el Desarrollo Armónico de la Región

Ajax / fetch / axios

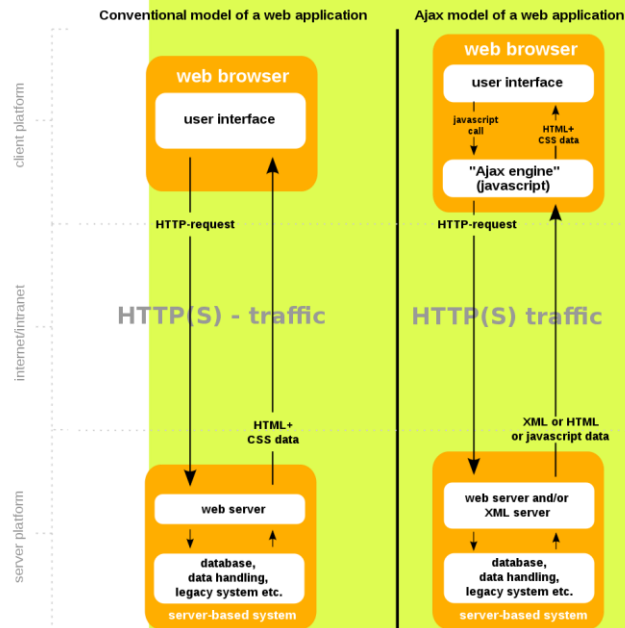
Ajax, Fetch y Axios

Ajax es una técnica de programación utilizada para construir páginas web más complejas y dinámicas, utilizando una tecnología conocida como XMLHttpRequest. Las siglas de Ajax responden a Asynchronous JavaScript and XML, o Javascript y XML Asíncrono.

Ajax permite modificar partes específicas del DOM de una página HTML sin la necesidad de refrescar la página entera. También permite trabajar asincrónicamente, lo que significa que el código seguirá corriendo mientras esa parte de la página de tu sitio web intenta recargarse.

Fetch es una API moderna que proporciona una forma más fácil y flexible de realizar solicitudes HTTP que Ajax.

Axios es una biblioteca de cliente HTTP para JavaScript que se basa en promesas para enviar peticiones HTTP.



¿Preguntas?



Universidad
Provincial del Sudoeste
Promoviendo el Desarrollo Armónico de la Región

Resumen de la clase hoy

- ✓ Introducción a la materia
- ✓ Primeros pasos con JavaScript

Muchas gracias.



Universidad
Provincial del Sudoeste
Promoviendo el Desarrollo Armónico de la Región