Asincronia Callbacks
Promesas
Async/Await
Api Rest

ASINCRONIA

Javascript es un lenguaje de programación asíncrono y no bloqueante con un manejador de eventos conocido como el **Events Loop** implementado en un único hilo para sus interfaces de entrada y salida.

Para entender mejor el concepto hay que imaginarse javascript como un aeropuerto, donde el events loop es la torre de control y el hilo principal es la pista de aterrizaje.

Un acción asíncrona es una acción que no ocurre al mismo tiempo de ejecución.

ASINCRONIA

Todas las tareas sincronas se ejecutan en el hilo principal de forma ordenadas mientras que las tareas asincronas se almacenan en una cola de tareas.

Cuando el hilo principal esté vació el events loop empezará a procesar las tareas asincronas de forma ordenada.

¿Cuales son las tareas asincronas?

- funciones asíncronas (setTimeout / setInterval)
- Callback
- Promesas

ASINCRONIA

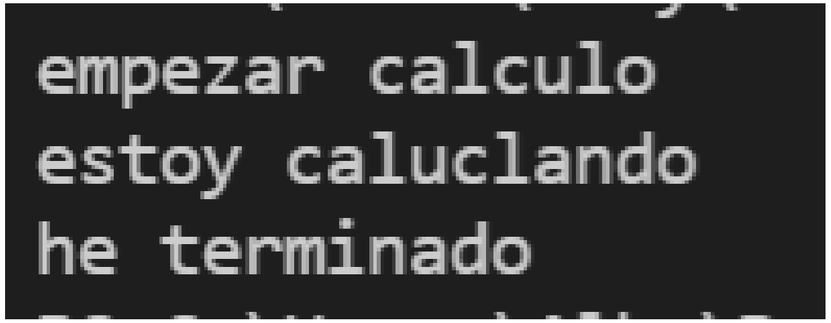
PROBLEMA

Imaginemos que tenemos una serie de instrucciones ordenadas en nuestro código y que simulemos a través de un ciclo for una tarea que tarde 5 segundos en ejecutarse

INPUT

```
console.log("empezar calculo");
for (let i = 0; i < 5; i++){
    console.log("estoy calculando");
}
console.log("calculo terminado");
```

OUTPUT



ASINCRONIA

PROBLEMA

Ahora vamos a modificar nuestro código y en lugar de utilizar un ciclo for vamos a utilizar la función asíncrona **setTimeout**

INPUT

```
console.log("empezar calculo");
setTimeout(() => {
    console.log("estoy calculando");
}, 5000);
console.log("calculo terminado");
```

OUTPUT

```
empezar calculo
calculo terminado
estoy calculando
```

ASINCRONIA

PROBLEMA

En el caso de utilizar setTimeout el orden de outputs no es el esperado ya que primero se ejecutan todas las tareas en el hilo principal y solamente cuando esté se quede libre se empezará a procesar las tareas asincronas.

CALLBACKS

Un callback (llamada de vuelta) es una función que recibe como argumento otra función y la ejecuta.

Nosotros ya hemos utilizado callbacks cuando signabamos un evento a un elemento del DOM.

```
let button = document.getElementById('boton');
button.addEventListener('click', function(){
    //Esta es mi funcion de callback
})
```

CALLBACKS

Los callbacks nos pueden ayudar a solucionar problemas de asincronia.
Vamos a ver como cambiaria nuestro código.

```
console.log("empezar calculo");

const endProcessData = () => {
    console.log("calculo terminado");
}

const processData = (callback) => {
    setTimeout(() => {
        console.log("estoy calculando");
        callback();
    }, 5000);
}

processData(endProcessData);
```

CALLBACKS

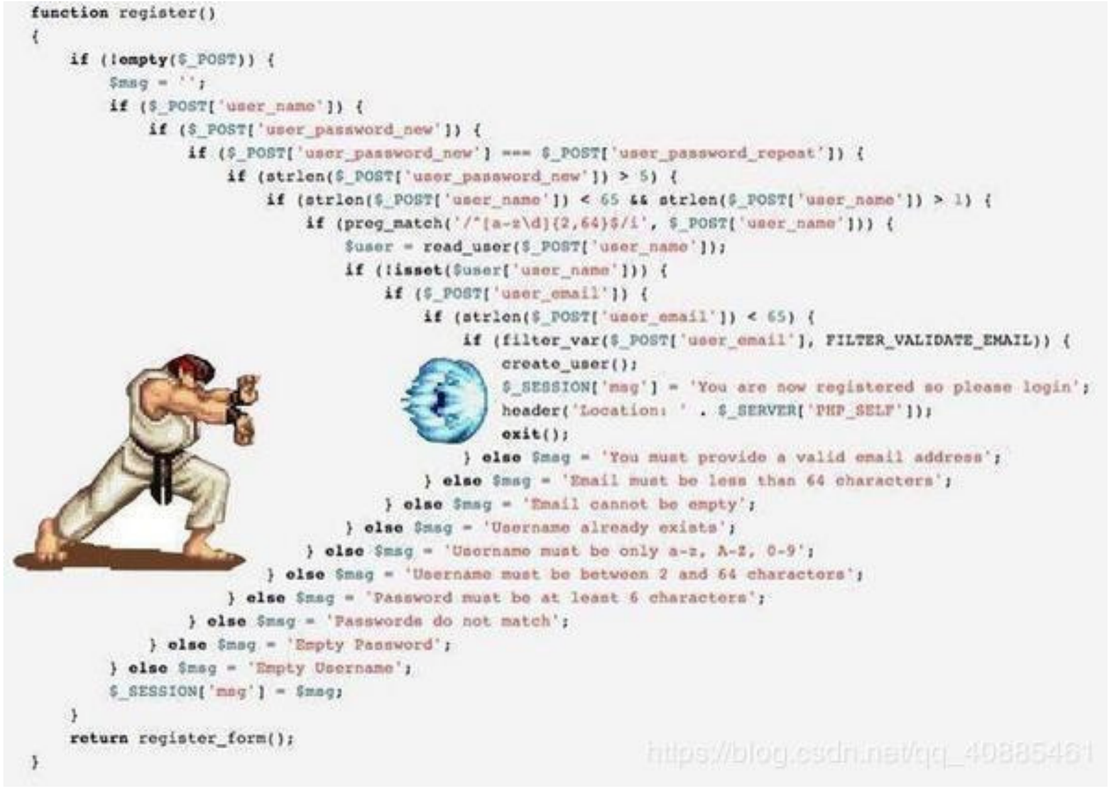
Podemos anidar tantos callback como sea necesario

```
const goToClass = () => {
    setTimeout(() => {
        console.log('Me levanto');
        setTimeout(() => {
            console.log('Desayuno');
            setTimeout(() => {
                console.log('Voy a clase')
            }, 700)
        }, 500)
    }, 200)
}

goToClass();
```

CALLBACKS

Anidar muchos callbacks no es una buena practica ya que puede llevarnos al problema del "**callbacks hell".** En su ugar es mejor utilizar **Promesas.**



PROMESAS

Una promesa es un objeto que representa un valor que puede que esté disponible «ahora», en un «futuro» o que «nunca» lo esté. Como no se sabe cuándo va a estar disponible, todas las operaciones dependientes de ese valor, tendrán que posponerse en el tiempo.

Vamos a ver como se declara una promesa

```
const p = new Promise((resolve, reject) =>{
    //Nuestra tarea asincrona
})
```

PROMESAS

- RESOLVE: es un callback que recibe nuestra promesa y que se ejecuta en el caso que la promesa consiga resolverse.
- REJECT: es un callback que recibe nuestra promesa y que se ejecuta cuando nuestra promesa no se cumple.

```
let x = 5;

const p = new Promise((resolve, reject) =>{
   if( x === 5){
    resolve('EL numero es correcto');
   }else{
    reject("El numero no es es correcto");
   }
})

p
.then(res => console.log(res))
.catch(err => console.error(err));
```

PROMESAS

Vamos a ver ahora como cambiaria nuestro primer ejemplo

```
console.log("Empiezo procesando los datos");
const p = new Promise((resolve, reject) =>{
    setTimeout(() => {
        console.log("Procesando los datos");
        resolve("Datos procesados");
    }, 200);
})

p
.then(res => console.log("El proceso ha finalizado correctamente"));
```

PROMESAS

Las promesas nos permiten encadenar tantas acciones cuanta sean necesarias

```
let x = 5;

const p = new Promise((resolve, reject) =>{
   if( x === 5){
    resolve('EL numero es correcto');
   }else{
    reject("El numero no es es correcto");
   }
})

p
.then(res =>  res)
.then(message => console.log(message))
.catch(err => console.error(err));
```

PETICIONES HTTP

Las peticiones HTTP son mensajes enviados por un cliente, para iniciar una acción en el servidor.

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.

PETICIONES HTTP

- **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.
- **PATCH:** El método PATCH es utilizado para aplicar modificaciones parciales a un recurso.
- **DELETE:** El método DELETE borra un recurso en específico.

PETICIONES HTTP

Las peticiones HTTP suelen estar compuestas por 4 elementos:

- El objetivo de una petición, normalmente es una URL, o la dirección completa del protocolo, puerto y dominio también suelen ser especificados por el contexto de la petición.
- Metodo: GET - POST - ETC..
- Cabecera
- Cuerpo

PETICIONES HTTP

HEADER HTTP

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 esta compuesta por su nombre (no sensible a las mayusculas) seguido de dos puntos ':', y a continuación su valor (sin saltos de línea). Los datos enviados más frequentemente son :

1. **Authorization**: permite autenticar una petición.
2. **Content-Type**: permite especificar el formato de datos que se envian

PETICIONES HTTP

BODY HTTP

La parte final de la petición es el cuerpo. No todas las peticiones llevan uno: las peticiones que reclaman datos, como GET, HEAD, DELETE, o OPTIONS, normalmente, no necesitan ningún cuerpo. Algunas peticiones pueden mandar peticiones al servidor con el fin de actualizarlo: como es el caso con la petición POST (que contiene datos de un formulario HTML).

PETICIONES HTTP

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.

```
var url = 'https://example.com/profile';
var data = {username: 'example'};

fetch(url, {
  method: 'POST',
  body: JSON.stringify(data),
  headers:{
    'Content-Type': 'application/json'
  }
}).then(res => res.json())
.then(response => console.log('Success:', response))
.catch(error => console.error('Error:', error));
```

ASYNC/AWAIT

En 2017 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.

```
const getData = async () => {
    let response = await fetch(url, {
        method: 'POST',
        body: JSON.stringify(data),
        headers:{
          'Content-Type': 'application/json'
        }
      });
    const data = await response.json();
    return data();
}

const info = getData();
```