

{desafío}
latam_

Callbacks y APIs _

Parte I



Introducción a la asincronía

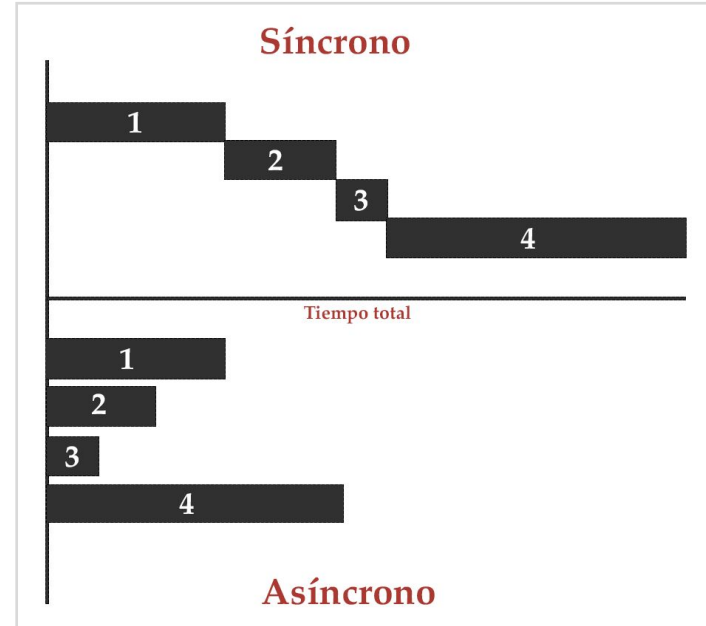
- Definir el término de asincronía en JavaScript.
- Describir el funcionamiento del Event Loop para entender la programación asíncrona.
- Comprender el concepto de Race Condition para observar el funcionamiento de la asincronía

Competencias

Sincronía y Asincronía

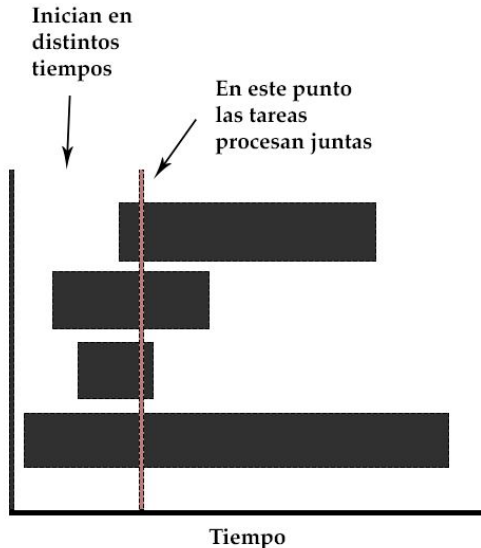
Síncrono, cuando los procesos se ejecutan uno tras otro.

Asíncrono, cuando los procesos se ejecutan todos a la vez y no necesitan esperar a que finalicen los otros.

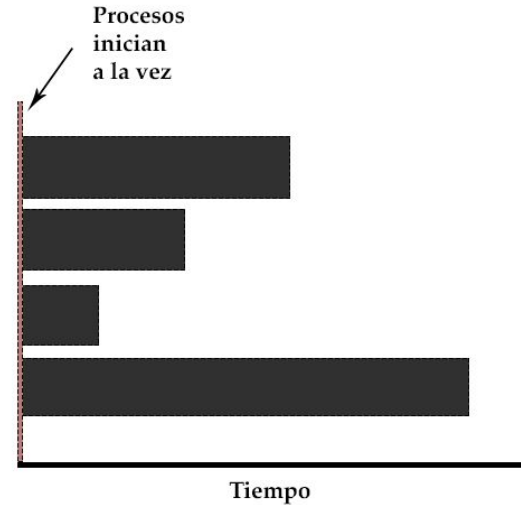


Procesos asíncronos

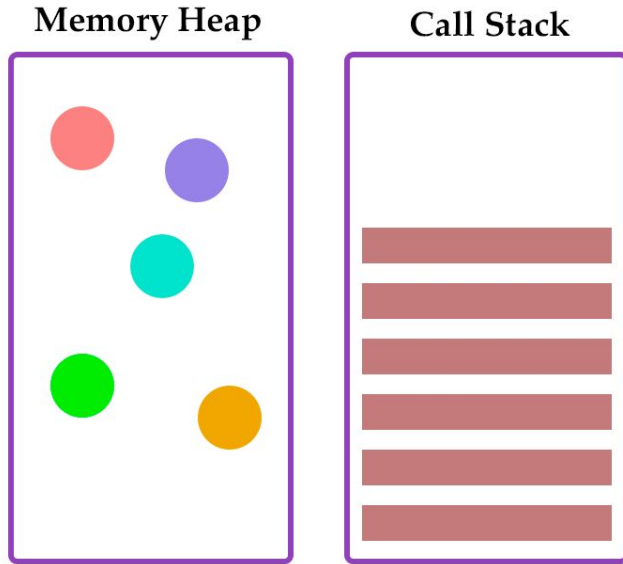
Concurrencia: Esto se produce cuando dos o más tareas progresan simultáneamente.



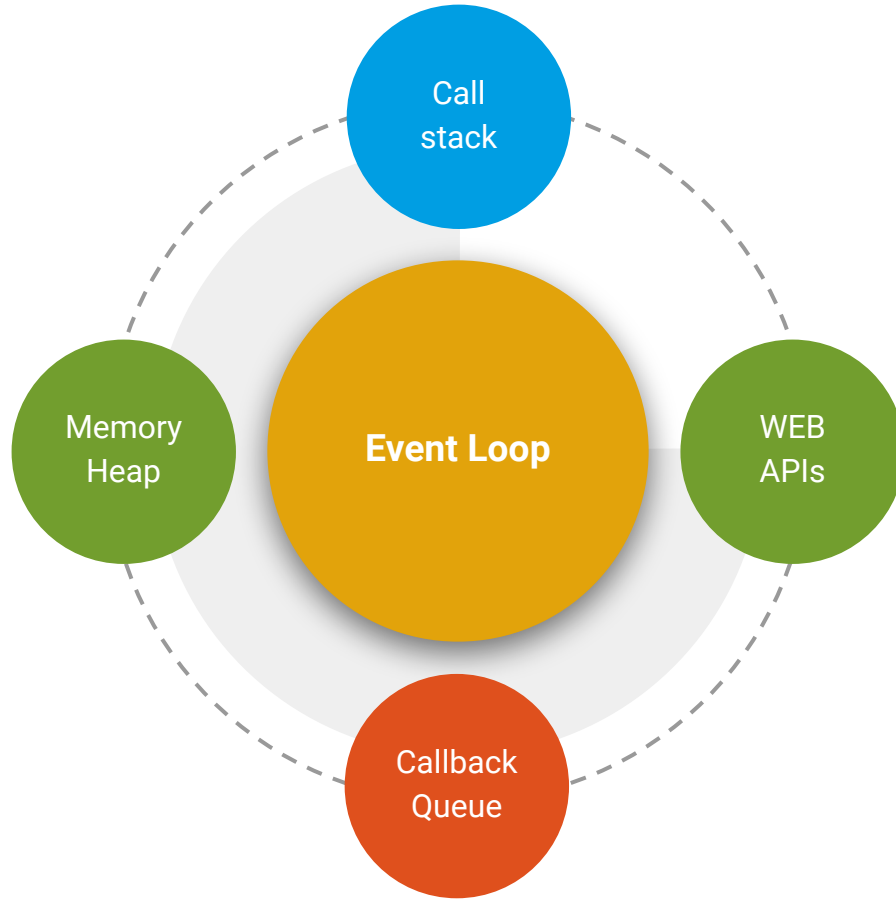
Paralelismo: Ocurre cuando dos o más se ejecutan, literalmente a la vez, en el mismo instante de tiempo.



Se encarga de implementar las operaciones asíncronas

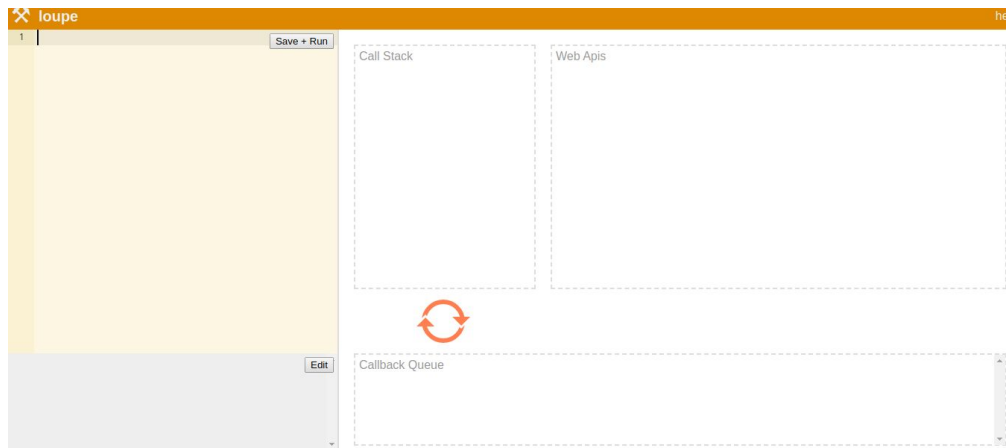


Event Loop o
Bucle de
Eventos



Event Loop o Bucle de Eventos

Utilizaremos una página que demuestra de manera gráfica los pasos que se van realizando desde que comienza la ejecución hasta que termina. Para ver lo que está ocurriendo utilizaremos el sitio web [Loupe](#) que es de propiedad de [Philip Roberts](#), aquí realizaremos nuestras pruebas.

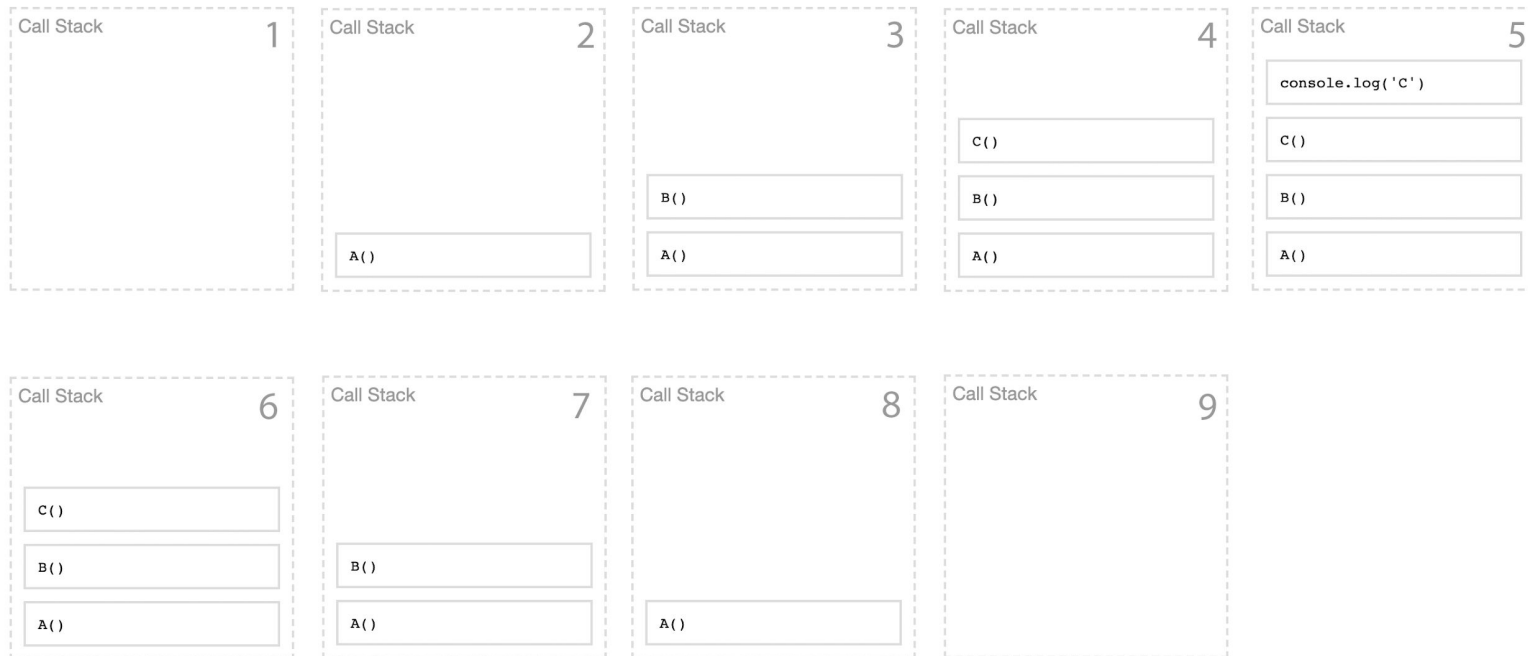


Ejercicio guiado: Callback Queue

Para mostrar el funcionamiento del **Callback Queue** se realizará un ejemplo implementado el sitio web [Loupe](#), más un código que tenga varias funciones anidadas y la última función muestre por la consola el literal “C”, una función debe llamar a la otra.

```
function C(){  
    console.log('C')  
}  
function B(){  
    C()  
}  
function A(){  
    B()  
}  
A();
```

Ejecución de la pila en orden



Ejercicio propuesto

Ejecuta el código mostrado en la página web [Loupe](#), y describe el comportamiento con tus propias palabras.

```
var A = [1, 2], B = [];  
  
for (var i = 0; i < A.length; i++) {  
    B.push(sumarDos(A[i]));  
};  
  
function sumarDos(x) {  
    return x + 2;  
};  
  
console.log(B)
```


Callbacks

- Una función que se pasa como argumento de otra función, y que será invocada para completar algún tipo de acción.
- Los callbacks se utilizan para esperar la ejecución de otros códigos antes de ejecutar el propio.

```
function foo(callback) {  
  //hacer algo...  
  callback();  
}
```

Crear una función que devuelva el nombre “John Doe”, el cual tiene que mostrarse en la consola del navegador y luego de ser obtenido, se debe implementar funciones con callback.

Realizar el ejercicio con ES6 y ES5 para visualizar mejor el callback y su funcionamiento.

A vertical decorative line on the right side of the slide, featuring a series of white and gray symbols: a closing curly brace '}', an '@' symbol, an opening curly brace '{', and a closing parenthesis ')'.

Ejercicio guiado: Implementación de Callbacks

Crear tres funciones que se utilizarán con un solo llamado, dos de ellas se ejecutarán dentro de la primera, y devolverán el resultado en la función callback. La primera función debe retornar el doble del argumento pasado, mientras que la segunda función debe retornar el cuadrado del argumento y por último la tercera función debe retornar el 25% del número.

Ejercicio guiado: Callbacks anidados

Desarrollar un programa con JavaScript implementando funciones con callback, que al pasarle los puntos obtenidos en dos ejercicios de un examen, sume ambos puntos e indique si hemos superado la prueba. El examen tiene una ponderación total de 10 puntos. Cada ejercicio tiene una nota máxima de 5 puntos.

Ejercicio propuesto

Race condition

Se interpreta como la ejecución de varios procesos o eventos a la vez modificando datos de forma concurrente, sin tener la certeza sobre cuáles serán los valores finales retornados por estos procesos, lo que puede producir errores o inconsistencia en los resultados esperados.

```
let value = 0;
function add1(callback) {
  callback(value += 1);
}
function add2(callback) {
  callback(value += 2);
}
```


Ejercicio guiado: Demostración de Race Condition

- Demostrar que en ocasiones los resultados pueden variar cuando el tiempo de ejecución influye en lo que queremos obtener.
- Crear una función que retorne un valor aleatorio entre dos números y esté utilizado para asignar diferentes tiempos de ejecución.
- Se debe implementar la función “setTimeout” para lograr que las funciones se ejecuten asíncronamente.

¿Cómo se puede evitar la condición de carrera?

1. Evitar utilizar recursos compartidos, como variables que se modifican en más de una función.
2. Tener cuidado con el alcance de las variables.
3. Utilizar correctamente la secuencia de instrucciones y verificar que el código se ejecuta de manera asíncrona, pero obteniendo los resultados de forma secuencial.
4. Nunca asumir que si se espera una cantidad de tiempo, el código se ejecutará en orden.

Crear la función "separar", donde se pasen dos (2) argumentos, un arreglo de números y un callback. La función deberá devolver un objeto con dos (2) arreglos, uno con los pares y otro con los impares. Ejemplo: Si se tiene el arreglo [3,12,7,1,2,9,18]. La función debe retornar: pares: [12,2,18], impares: [3,7,1,9].

Ejercicio propuesto

Callbacks y Promesas

- Codificar una función asíncrona para utilizar callbacks y setTimeout.
- Codificar una función asíncrona que permita implementar Promise / Resolve.

Competencias

setTimeout

El método `setTimeout()` llama a una función o evalúa una expresión después de un número específico de milisegundos.

Funcionalidades:

El callback pasado como primer argumento se ejecutará después del tiempo establecido en el segundo argumento.

Su ejecución no bloquea el stack, por lo que es una función que se procesa de forma asíncrona.

setTimeout

Sintaxis:

- El único argumento requerido es "function".

```
setTimeout(function(){}, milliseconds, param1, param2, ...)
```

Implementación de setTimeout

El primer ejemplo consiste en esperar una cantidad de tiempo para evaluar la expresión que contiene la función que se le pasó por argumento, que en este caso es mostrar en la consola un mensaje.

```
setTimeout(() => {  
    console.log('hola mundo!');  
}, 1000)
```


Simular una petición asíncrona como si se tratara de una API, retornando la información de un usuario pero en formato JSON. La información para el primer usuario será la siguiente: `data = {id: 1,name: 'John',lastName: 'Doe',age: 24}`. Mientras que la información para el segundo usuario será: `data = {id: 2,name: 'Jane',lastName: 'Smith',age: 19}`.

Por lo tanto, se debe enviar el parámetro “id” a la función para simular que se requieren los datos de un usuario en específico y que el tiempo de respuesta será de 1000 milisegundos.

Ejercicio guiado: Obteniendo usuarios por ID

Ejercicio propuesto

Crear un programa con JavaScript mediante el uso de funciones con Callback y el método `setTimeout`, que permita mostrar los datos de un usuario de acuerdo al nombre o el número de identificación. Los datos serán los siguientes:

```
usuario1 = {id: 2356256,name: 'Juan',lastName: 'Duran',age: 35}  
usuario2 = {id: 27564512,name: 'Manuel',lastName: 'Perez',age: 31}  
usuario3 = {id: 17658624,name: 'Jocelyn',lastName: 'Rodriguez',age: 30}  
usuario4 = {id: 12345678,name: 'Maria',lastName: 'Garrido',age: 30}
```

Promesas (Promise)

Una promesa es un objeto que representa el estado de una operación asíncrona. Estos estados son:

- **pendiente** - cuando todavía no empieza o no ha terminado de ejecutarse.
- **resuelto** - en caso de éxito.
- **rechazado** - en caso de fallo.

```
new Promise((resolve, reject) => { /* ..... */ })
```

Promesas (Promise)

Se requiere pasar como argumento una función callback que contenga dos (2) parámetros, puedes nombrarlos como quieras, en este caso se usarán `resolve` y `reject` que son los valores por defecto. Como se muestra a continuación:

```
const promise = new Promise((resolve, reject) => {  
  const value = true;  
  value ? resolve('Exito') : reject('Rechazado')  
})  
promise.then(resp => console.log(resp)) // output: Exito
```

Exito

VM21729:6

◀ ▶ Promise {<resolved>: undefined}

Promise.all

- Permite ejecutar un objeto iterable o arreglo de múltiples promesas, y esperar que se resuelvan para entregar todos los resultados de una vez.
- “Retorna” es un arreglo con tantos elementos que se le hayan pasado al método *all* en el objeto iterable.

```
Promise.all(objetoIterable)
  .then((result) => { /* ... */ })
  .catch((error) => { /* ... */ })
```

Aplicando la sintaxis anterior, se deben cumplir todas las promesas iniciadas en variables separadas, la primera se debe igualar a un número, la segunda se debe igualar a `Promise.resolve(2)`, y la tercera a una nueva promesa que retorne el número tres cuando se resuelva.

Implementar la sentencia del `Promesa.all` y ver cómo se deben cumplir todas las promesas para que se retorne un arreglo con los resultados.

Ejercicio guiado: `Promise.all`

Ejercicio guiado: Combinando promesas y eventos del DOM

- Solicitan que al hacer un clic sobre un botón, se ejecute una promesa que retorne y muestre por pantalla tres mensajes:
 - el primero: “Solicitando Autorización”
 - el segundo: “Esperando la información”
 - el tercero: “El usuario en línea es: Juan”.

Estos mensajes se deben insertar en el documento utilizando el DOM dentro de una función externa.

- Para generar el primer mensaje se debe utilizar una función que contenga una promesa y tarde 2.5 segundos en indicar que todo está correcto, es decir, que tiene autorización retornando “true”, mientras que el segundo y tercer mensaje se origina en otra función que contiene una promesa que muestra primeramente el segundo mensaje y al cabo de 2,5 segundos muestra el tercer y último mensaje.

Crear un programa con JavaScript utilizando promesas que calcule un número aleatorio entre el “1” y el “100”, pero que muestre el número aleatorio si y sólo si este número está comprendido entre “20” y “60”, ambos valores incluidos.


Ejercicio propuesto

Realizar un ejemplo donde existan tres promesas, cada una regresará un valor distinto (1, 2 y 3 respectivamente), de acuerdo a la ejecución del tiempo que tenga cada una.

Ese tiempo debe originarse en una función externa, la cual va a retornar un número aleatorio que servirá como tiempo para los `setTimeout` de cada promesa, dando un valor aleatorio a cada una de ellas.

Ejercicio guiado: Promise.race

En un colegio, la profesora decide dar incentivos a quien responda primero las preguntas que hace. El que lo haga obtendrá décimas para la próxima evaluación. Selecciona a Carlos, María y Cristian para que respondan. Los estudiantes piensan y dan su respuesta casi al mismo tiempo, ¿quién da la respuesta más rápido?

A vertical line separates the white text area from the blue text area. It features a series of white symbols: a closing curly brace '}', an at-sign '@', an opening curly brace '{', and a stylized person icon, all arranged vertically.

**Ejercicio
guiado:
Promise.race,
la respuesta
más rápida**

Realizar un programa en JavaScript que calcule la suma o la resta o la multiplicación de dos números. Para ello se debe implementar una promesa por cada operación matemática y solo se debe mostrar el resultado de la promesa ganadora. Utiliza tiempos aleatorios para los `setTimeout` que poseen cada promesa con su operación matemática. Igualmente implementa `Promise.race` para mostrar sola la primera promesa que retorne el resultado de de la operación matemática ganadora.

Ejercicio propuesto

Async/Await en JavaScript

- Codificar una función asíncrona utilizando ASYNC/AWAIT para obtener una respuesta directa y no una promesa.
- Identificar las restricciones de Async / Await en ES6 para evitar errores de implementación en el código

Competencias

Async / Await

- Permite que las funciones que retornan promesas, devuelvan directamente los resultados en vez de promesas.
- Se utiliza la palabra clave `async` antes de la declaración de una función, lo que nos indica que siempre retornará una promesa.
- **Sintaxis Async:**

```
async function name([param[, param[, ... param]]) { /* ... */ }
```


Async / Await

La palabra reservada `await` la utilizaremos para esperar y retornar la promesa. El operador `await` solo trabaja dentro de una función `async`.

Sintaxis Await

```
let value = await promise;
```

Crear un programa que consulte una URL que entregue fotos aleatorias de perros, siendo esta URL: <https://dog.ceo/api/breeds/image/random>. Además se debe implementar Async / Await a lo largo del ejercicio y utilizar la estructura [try...catch](#). Para ejecutar el código y atrapar un error en el caso de existir uno. Se solicita implementar el método fetch.

A vertical line separating the text from the title, featuring a large '@' symbol and curly braces '{' and '}' in white and light gray.

Ejercicio guiado: Controlando la asincronía

Restricciones

Cualquier función puede ser asíncrona y utilizar **async**, pero hay que tener en cuenta que la función pasa a retornar una promesa, por lo que debe continuar su ejecución con **.then** para obtener el resultado.

Para utilizar la palabra reservada **await** debe hacerse solamente en funciones declaradas con **async**. No se puede usar en funciones regulares.

Hay que tener cuidado con el alcance de las funciones asíncronas, ya que por su naturaleza dejan continuar con la ejecución del resto del código.

Ejercicio propuesto

Convertir el siguiente código para que se pueda usar promesas.

```
const log = (text, callback) => {
  setTimeout(() => {
    console.log(`${text}`);
    callback()
  }, 1000)
}

log('uno', () => {
  log('dos', () => {
    log('tres', () => { })
  })
})
```

Crea una función que permita traer y mostrar por la consola del navegador web la información de la siguiente dirección web:

<https://www.feriadosapp.com/api/holidays.json>.

implementa Async...Await

Ejercicio propuesto

{desafío}
latam_

*Academia de
talentos digitales*

www.desafiolatam.com