Programacion asicrona

Temporizadores setTimeout y setInterval

setTimeout

El setTimeout ese una función de javascript que sirve para ejecutar una función tras un periodo determinado cabe recalcar que esta función solo se ejecuta una sola vez pasado el tiempo dado.

console.log("Inicio");

// Ejecutar una función después de 3 segundos

setTimeout(() => {

console.log("Mensaje después de 3 segundos");

}, 3000);

console.log("Fin");

Cleartimeout

La función cleartimeout lo que hace es borrar (stop) el tiempo de espera de él settimeout

console.log("Inicio");

// Inicia un temporizador que se ejecuta después de 5 segundos

const timeoutId = setTimeout(() => {

console.log("Este mensaje no debería aparecer");

}, 5000);

// Cancela el temporizador después de 2 segundos

setTimeout(() => {

clearTimeout(timeoutId);

console.log("Temporizador cancelado");

}, 2000);

console.log("Fin");

ClearInterval

La función clearInterval lo que hace es parar o borrar el tiempo de setInterval

console.log("Inicio");

let contador = 0;

// Inicia un intervalo que se ejecuta cada segundo

const intervalId = setInterval(() => {

contador += 1;

console.log(`Contador: ${contador}`);

// Detiene el intervalo después de que el contador alcanza 5

if (contador >= 5) {

clearInterval(intervalId);

console.log("Intervalo detenido");

}

}, 1000);

console.log("Fin");

setInterval

Usa setInterval() para hacer que una función se repita con un tiempo de retraso entre cada ejecución.

Todos los argumentos tienen el mismo significado. Pero a diferencia de setTimeout, ejecuta la función no solo una vez, sino regularmente después del intervalo de tiempo dado.

console.log("Inicio");

const intervalId = setInterval(() => {

console.log("Mensaje repetido cada segundo");

}, 1000);

// Después de 5 segundos, detenemos el intervalo

setTimeout(() => {

clearInterval(intervalId);

console.log("Intervalo detenido");

}, 5000);

console.log("Fin");

Asincronía y el event loop

Asincronía

La programación asíncrona es una forma de que el programa inicie una tarea larga duración y siga respondiendo a otras tareas sin que el programa se congele a esperar que la tarea acabe para empezar otra.

console.log("Inicio");

setTimeout(() => {

console.log("Operación asíncrona completada");

}, 1000);

console.log("Fin");

Este es un ejemplo de programación asíncrona, ya que se sigue ejecutando el código sin esperar a que el console.log(“ejecutando”) termine el tiempo para poder mostrar el finalizado

Event loop

console.log("Inicio"); // Tarea síncrona, se ejecuta inmediatamente

setTimeout(() => {

console.log("Timeout 1"); // Tarea de la cola de macrotareas, se ejecuta después de todas las tareas síncronas y promesas resueltas

}, 0);

Promise.resolve().then(() => {

console.log("Promesa 1"); // Tarea de la cola de microtareas, se ejecuta después de todas las tareas síncronas pero antes de las macrotareas

});

Promise.resolve().then(() => {

console.log("Promesa 2"); // Otra tarea de la cola de microtareas, se ejecuta después de todas las tareas síncronas pero antes de las macrotareas

});

setTimeout(() => {

console.log("Timeout 2"); // Otra tarea de la cola de macrotareas

}, 0);

console.log("Fin"); // Otra tarea síncrona, se ejecuta inmediatamente después de "Inicio"

call back

Una función de callback es una función que se pasa a otra función como un argumento

// Función que acepta un callback

function processData(callback) {

const data = "Datos procesados";

callback(data);

}

// Definimos el callback

function displayData(data) {

console.log(data);

}

// Llamamos a la función con el callback

processData(displayData);

// Output: Datos procesados

funciones callback con listas

// Array de números

const numbers = [1, 2, 3, 4, 5];

// Definimos el callback para el método map

function square(num) {

return num \* num;

}

// Usamos el callback con el método map

const squaredNumbers = numbers.map(square);

console.log(squaredNumbers);

// Output: [1, 4, 9, 16, 25]

promesas

Las promesas es algo que puede ocurrir y se puede cumplir, como tampoco se podrá cumplir

**Las promesas tienes como 3 estados**

-La promesa que se cumple

-La promesa que no se cumple

-La promesa se queda en un estado pendiente, como esperando las respuesta

**Métodos para las promesas**

**.then =** ejecuta la función callback cuando la promesa se cumple

**.catch =** ejecuta la función callback cuando la promesa se rechaza

**.finally =** ejecuta la función callback tanto si se cumple o como si se rechaza

// Función que devuelve una promesa

function fetchData() {

return new Promise((resolve, reject) => {

setTimeout(() => {

const success = true; // Cambia esto a false para simular un error

if (success) {

resolve("Datos recibidos");

} else {

reject("Error al obtener los datos");

}

}, 2000);

});

}

// Llamada a la función que devuelve una promesa

fetchData()

.then(data => {

console.log(data); // "Datos recibidos"

})

.catch(error => {

console.error(error); // "Error al obtener los datos"

});

funciones asíncronas, async await

Tenemos dos partes cuando usamos async/await en nuestro código.

En primer lugar, tenemos la palabra clave async , que se pone delante de una declaración de función para convertirla en una función async.

Una función asíncrona es una función que sabe que es posible que se use la palabra clave await dentro de ella para invocar código asíncrono.

La palabra clave async se añade a las funciones para que devuelvan una promesa en lugar de un valor directamente.

// Función que simula una tarea asíncrona

function delay(ms) {

return new Promise(resolve => setTimeout(resolve, ms));

}

// Función asíncrona que utiliza 'await' para esperar la finalización de la tarea

async function exampleFunction() {

console.log("Inicio de la tarea...");

await delay(2000); // Espera 2 segundos

console.log("Tarea completada después de 2 segundos");

}

// Llamada a la función asíncrona

exampleFunction();