COLOMBIA POTENCIA DE LA VIDA

TIC

TALENTO TECH

DESARROLLO *WEB*

FULL STACK

NIVEL BÁSICO

Programación Asíncrona en JavaScript – Callbacks, Promesas y Async/Await

cymetria | tecnalia colombia

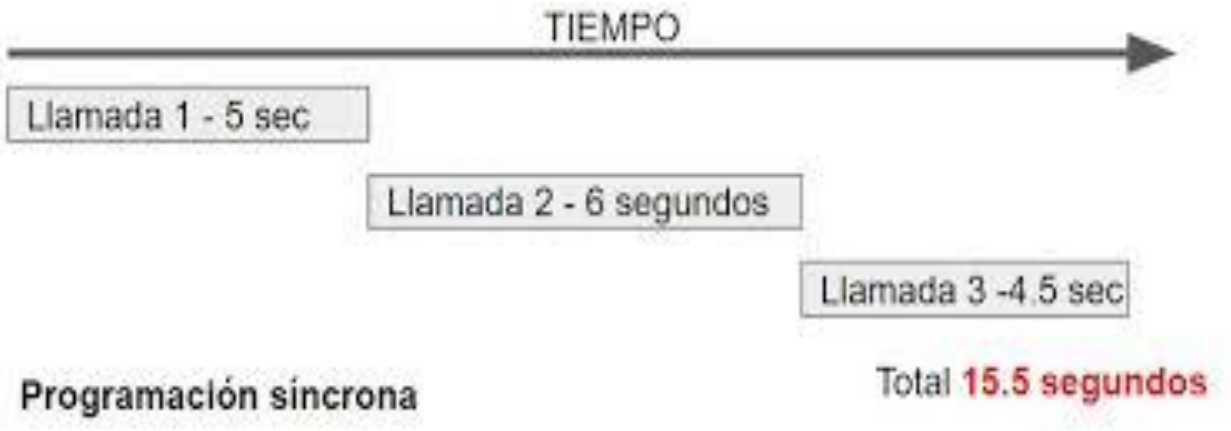
Introducción a la Programación Asíncrona

JavaScript es un lenguaje **single-threaded**, lo que significa que ejecuta una tarea a la vez en un solo hilo. Sin embargo, muchas veces necesitamos realizar operaciones que pueden tomar tiempo, como:

- **Peticiones a una API o base de datos**
- **Leer archivos desde el sistema**
- **Esperar la respuesta de un servidor**
- **Retrasar la ejecución de un código**

"Single-threaded" o "de un solo subproceso" se refiere a un modelo que permite ejecutar una sola tarea a la vez

Si JavaScript esperara a que estas tareas terminen antes de seguir ejecutando el código, el navegador se quedaría bloqueado. Aquí es donde entra la **programación asíncrona**, que permite manejar tareas sin bloquear la ejecución principal.



Callbacks: La Forma Más Antigua(Código Asíncrono)

Un **callback** es una función que se pasa como argumento a otra función y se ejecuta después de que se complete una operación, generalmente asíncrona (como una solicitud a una API o la lectura de un archivo).

¿Por qué usamos callbacks?
En JavaScript, muchas operaciones son asíncronas (no ocurren inmediatamente). Los callbacks nos permiten manejar estas operaciones sin bloquear el flujo del programa.

Obtener datos de usuarios desde una API
Imaginemos que estamos construyendo una aplicación que necesita obtener información de usuarios desde una API

En el pasado, los callbacks eran la forma principal de manejar operaciones asíncronas en JavaScript. Sin embargo, cuando hay muchas operaciones anidadas, el código se vuelve difícil de leer (el famoso "Callback Hell").

Callbacks: La Forma Más Antigua

Supongamos que queremos:

1.Obtener un usuario desde una API.

2.Luego, obtener los posts

3.de ese usuario.

4.Finalmente, mostrar los los

posts en la consola.

Problema:

Si tuviéramos más operaciones anidadas, el código se volvería difícil de leer y mantener. Esto se conoce como Callback Hell.

```
// Simulamos una función que hace una solicitud a una API (con setTimeout)
function obtenerUsuario(id, callback) {
    setTimeout(() => {
        const usuario = { id: id, nombre: "Juan" }; // Simulamos un usuario
        callback(usuario);
    }, 1000); // Simulamos un retraso de 1 segundo
}

function obtenerPosts(usuario, callback) {
    setTimeout(() => {
        const posts = ["Post 1", "Post 2", "Post 3"]; // Simulamos los posts
        callback(posts);
    }, 1000); // Simulamos otro retraso de 1 segundo
}

// Usamos callbacks para manejar las operaciones asincronas
obtenerUsuario(1, (usuario) => {
    console.log("Usuario obtenido:", usuario);
    obtenerPosts(usuario, (posts) => {
        console.log("Posts del usuario:", posts);
    });
});
```

Callbacks: La Forma Más Antigua

Problema con los Callbacks: Anidar múltiples callbacks puede llevar a lo que se conoce como Callback Hell, haciendo que el código sea difícil de leer y mantener.

javascript

```javascript
function paso1(callback) {
    setTimeout(() => {
        console.log("Paso 1 completado");
        callback();
    }, 1000);
}

function paso2(callback) {
    setTimeout(() => {
        console.log("Paso 2 completado");
        callback();
    }, 1000);
}

paso1(() => {
    paso2(() => {
        console.log("Todos los pasos completados");
    });
});
```

▼ Callback Hell ocurre cuando hay demasiados niveles anidados.

Promesas: Una Mejor Forma de Manejar Código Asíncrono

Una **promesa** es un objeto que representa el resultado de una operación asíncrona. Puede estar en uno de tres estados:

1. **Pendiente (Pending):** La operación aún no se ha completado.
2. **Cumplida (Fulfilled):** La operación se completó con éxito.
3. **Rechazada (Rejected):** La operación falló.

Las promesas permiten manejar operaciones asíncronas de manera más limpia y evitar el "Callback Hell".

Encadenando Promesas (Evita Callback Hell)

```
// Simulamos una función que devuelve una promesa para obtener un usuario
function obtenerUsuario(id) {
    return new Promise((resolve) => {
        setTimeout(() => {
            const usuario = { id: id, nombre: "Juan" }; // Simulamos un usuario
            resolve(usuario);
        }, 1000); // Simulamos un retraso de 1 segundo
    });
}

// Simulamos una función que devuelve una promesa para obtener los posts
function obtenerPosts(usuario) {
    return new Promise((resolve) => {
        setTimeout(() => {
            const posts = ["Post 1", "Post 2", "Post 3"]; // Simulamos los posts
            resolve(posts);
        }, 1000); // Simulamos otro retraso de 1 segundo
    });
}

// Usamos promesas para manejar las operaciones asíncronas
obtenerUsuario(1)
    .then((usuario) => {
        console.log("Usuario obtenido:", usuario);
        return obtenerPosts(usuario); // Encadenamos la siguiente promesa
    })
    .then((posts) => {
        console.log("Posts del usuario:", posts);
    })
    .catch((error) => {
        console.error("Algo salió mal:", error);
    });
```

Async/Await

Async/await es una forma moderna y más legible de trabajar con promesas. La palabra clave async se usa para declarar una función asíncrona, y await se usa para esperar a que una promesa se resuelva.
¿Por qué usamos async/await?
Hace que el código asíncrono parezca síncrono, lo que facilita su lectura y mantenimiento.
Es más intuitivo que usar .then() y .catch().

Ventaja:
El código es mucho más legible y fácil de entender. Parece código síncrono, pero en realidad es asíncrono.

Resumen con el Ejemplo Real:

1. **Callbacks:**
   - Funciona, pero puede volverse complicado con muchas operaciones anidadas.
   - Ejemplo: Obtener usuario → Obtener posts → Mostrar posts.
2. **Promesas:**
   - Mejora la legibilidad y manejo de errores.
   - Ejemplo: Encadenar `.then()` para manejar operaciones secuenciales.
3. **Async/Await:**
   - Hace que el código sea más limpio y fácil de leer.
   - Ejemplo: Usar `await` para esperar a que las promesas se resuelvan.

Async/Await

```
// Usamos las mismas funciones de promesas que definimos antes
function obtenerUsuario(id) {
    return new Promise((resolve) => {
        setTimeout(() => {
            const usuario = { id: id, nombre: "Juan" }; // Simulamos un usuari
            resolve(usuario);
        }, 1000); // Simulamos un retraso de 1 segundo
    });
}

function obtenerPosts(usuario) {
    return new Promise((resolve) => {
        setTimeout(() => {
            const posts = ["Post 1", "Post 2", "Post 3"]; // Simulamos los posts
            resolve(posts);
        }, 1000); // Simulamos otro retraso de 1 segundo
    });
}

// Función asíncrona que usa async/await
async function mostrarDatosUsuario() {
    try {
        const usuario = await obtenerUsuario(1); // Esperamos a que se resuelva la promesa
        console.log("Usuario obtenido:", usuario);

        const posts = await obtenerPosts(usuario); // Esperamos a que se resuelva la siguiente p
romesa
        console.log("Posts del usuario:", posts);
    } catch (error) {
        console.error("Algo salió mal:", error);
    }
}

// Llamamos a la función asincrona
mostrarDatosUsuario();
```

Casos de Uso en Aplicaciones Web

🔹 Llamadas a API (Ejemplo con `fetch`)

javascript

Copy Edit

```javascript
async function obtenerPosts() {
    try {
        let respuesta = await fetch("https://jsonplaceholder.typicode.com/posts");
        let posts = await respuesta.json();
        console.log(posts);
    } catch (error) {
        console.error("Error al obtener los posts:", error);
    }
}

obtenerPosts();
```

Casos de Uso en Aplicaciones Web

Lectura de Archivos (Node.js)

javascript

Copy Edit

```javascript
const fs = require('fs').promises;

async function leerArchivo() {
    try {
        let contenido = await fs.readFile("archivo.txt", "utf8");
        console.log("Contenido:", contenido);
    } catch (error) {
        console.error("Error al leer el archivo:", error);
    }
}

leerArchivo();
```

Casos de Uso en Aplicaciones Web

Simulación de una Base de Datos

javascript Copy Edit

```javascript
function obtenerDatosDB() {
    return new Promise(resolve => {
        setTimeout(() => resolve(["Usuario1", "Usuario2", "Usuario3"]), 3000);
    });
}

async function mostrarUsuarios() {
    console.log("Cargando usuarios...");
    let usuarios = await obtenerDatosDB();
    console.log("Usuarios:", usuarios);
}

mostrarUsuarios();
```

Ejemplo de la Vida Real: Pizzería

Imaginemos que vamos a una pizzería y ordenamos una pizza. El proceso tiene varias etapas:

1. Tomar el pedido
2. Preparar la pizza
3. Hornear la pizza
4. Entregar la pizza

Cada una de estas etapas **toma tiempo**, pero el cajero no puede quedarse esperando sin hacer nada.

Necesita seguir atendiendo a otros clientes.

Vamos a ver cómo se maneja este proceso usando **callbacks**, **promesas** y **async/await**.

📁 Estructura de Archivos

```
/pizzeria/
|— index.html      (Estructura de la página)
|— style.css       (Estilos)
|— script.js       (Lógica del proceso de pedido)
```

Ejemplo de la Vida Real: Pizzería

📌 Descripción del Proyecto

1. Un botón en la página permite realizar el pedido.
2. Se simulan tres procesos asíncronos:
   - Tomar el pedido (Callback)
   - Preparar la pizza (Promesa)
   - Entregar la pizza (Async/Await)
3. Se actualiza la interfaz en lugar de mostrarlo solo en la consola.

📌 ¿Cómo funciona?

1. **El usuario hace clic en el botón "Pedir Pizza"**
   - Se inicia la función `iniciarPedido()`.
2. **Paso 1: Se toma el pedido (Callback)**
   - Se actualiza la interfaz a `"📞 Tomando el pedido..."`.
   - Luego de 2 segundos, se muestra `"✅ Pedido tomado"`.
3. **Paso 2: Se prepara la pizza (Promesa)**
   - Se actualiza la interfaz a `"👨‍🍳 Preparando la pizza..."`.
   - Después de 3 segundos, se muestra `"✅ Pizza lista"`.
4. **Paso 3: Se entrega la pizza (Async/Await)**
   - Se actualiza la interfaz a `"🚗 Repartidor en camino..."`.
   - Después de 2 segundos, se muestra `"✅ Pizza entregada 🍕"`.