<codoacodo/>

Agencia de Aprendizaje a lo largo de la vida

FULL STACK FRONTEND
Clase 20

Javascript 8

BA

Fundamentos del asincronismo en Javascript (Parte II)

Les damos la bienvenida

Vamos a comenzar a grabar la clase

Clase 18

DOM y Eventos

- Manipulación del DOM.
- Definición, alcance y su importancia..
- Eventos en JS.
- Eventos. ¿Qué son, para qué sirven y cuáles son los más comunes?
- Escuchar un evento sobre el DOM.

Clase 19

Asincronismo (Parte I)

- Introducción
- Sincronismo y Asincronismo
- Promesas
- Funciones del constructor y métodos.

Clase 20

Asincronismo (Parte II)

- Introducción
- async / await
- Manejo de errores con try/catch en funciones asíncronas
- Callbacks / Callback hell

Recordemos el concepto de asincronismo

Es la capacidad de ejecutar ciertas operaciones en ***segundo plano*** y continuar realizando otras tareas ***sin tener que esperar a que la operación asincrónica termine.***

En JavaScript, esto es esencial debido a que es un lenguaje de programación de un solo hilo, lo que significa que puede ejecutar un solo bloque de código a la vez. El asincronismo permite que JavaScript maneje tareas como cargar datos, procesar archivos grandes, o realizar solicitudes de ***red sin bloquear la interfaz de usuario o detener otras operaciones***. Utiliza funciones como ***callbacks, promesas y async/await*** para gestionar comportamientos asíncronos, permitiendo que las aplicaciones sean más rápidas y responsivas.

async / await

async y **await** son extensiones de las promesas en JavaScript que simplifican la forma de trabajar **con operaciones asíncronas.** Estos hacen que el código asíncrono sea más fácil de escribir y entender sin las complejidades habituales asociadas con el manejo de promesas.

La función async

La palabra clave async se utiliza para declarar una función como asíncrona. Declara que una función puede, en algún momento, esperar a que las promesas se resuelvan sin bloquear la ejecución de otras operaciones. Aquí están los detalles clave:

Devolución de una promesa: Una función async automáticamente devuelve una promesa. Si la función retorna algo que no es una promesa, ese valor será automáticamente envuelto en una promesa.

Sintaxis: Puede ser usada con declaraciones de funciones normales, funciones flecha, y métodos de clase.

async / await

En el siguiente ejemplo, declaramos una función asincrónica, por lo tanto devolverá automáticamente una promesa. Esto nos permite utilizar los métodos aplicados a las promesas cuando sea invocada la función.

```
async function obtenerDatos() {
    return "Datos recibidos";
}

// La función devuelve una promesa
obtenerDatos().then(console.log); // Imprime "Datos recibidos"
```

Pero todavía, al no utilizar el **await** dentro de la función, no tenemos un asincronismo real en el código.

async / await

La palabra clave await

await sólo puede ser usado dentro de funciones **async** y es usado para pausar la ejecución de la función hasta que la promesa a la que se llama se resuelva o rechace. El uso de await ayuda a que el código asíncrono se lea de manera más intuitiva, casi como si fuera sincrónico.

```
async function procesarDatos() {
    try {
        // Espera aquí hasta que fetch complete
        const datos = await fetch('https://api.example.com/datos');
        // Espera aquí hasta que la conversión a JSON complete
        const json = await datos.json();
        console.log(json);
    } catch (error) {
        console.error("Error al obtener datos:", error);
    }
}
```

async / await

Beneficios del uso de Async/Await

Claridad y Legibilidad: Al eliminar la necesidad de encadenar .then() y .catch(), el código se vuelve más claro y fácil de seguir.

Error Handling: El manejo de errores se simplifica mediante el uso de bloques try/catch, que son ya familiares para muchos desarrolladores y aplican de igual manera en contexto síncrono y asíncrono.

Depuración Simplificada: La depuración de código asíncrono puede ser complicada debido a las cadenas de promesas. Con async/await, puedes usar puntos de interrupción de depuración de manera más efectiva, ya que el flujo del código es más directo y predecible.

async / await

Consideraciones

A pesar de sus ventajas, **async/await** no reemplaza completamente a las promesas, especialmente en situaciones donde se requiere manejar múltiples promesas concurrentes. En esos casos, funciones como **Promise.all()** siguen siendo de gran utilidad para optimizar el rendimiento al permitir la ejecución paralela.

En conclusión, **async/await** es una herramienta poderosa en JavaScript que, cuando se usa adecuadamente, puede hacer que el manejo de la asincronía no sólo sea más manejable, sino también mucho más elegante y eficiente.

Manejo de Errores con Try/Catch

En síncrono: try/catch permite capturar errores en bloques de código específicos para evitar que los errores no controlados rompan el flujo de ejecución.

```
try {
    let resultado = operacionRiesgosa();
    console.log(resultado);
} catch (error) {
    console.error("Se capturó un error:", error);
}
```

Manejo de Errores con Try/Catch

En asíncrono: try/catch dentro de funciones async para manejar errores de promesas.

```
async function processData() {
    try {
        let data = await
fetch('https://api.example.com/data').then(res => res.json());
        console.log(data);
    } catch (error) {
        console.error("Error procesando los datos:", error);
    }
}

processData();
```

Manejo de Errores con Try/Catch

En asíncrono: try/catch dentro de funciones async para manejar errores de promesas.

```
async function processData() {
    try {
        let data = await
fetch('https://api.example.com/data').then(res => res.json());
        console.log(data);
    } catch (error) {
        console.error("Error procesando los datos:", error);
    }
}

processData();
```

Callbacks

Un **callback** es una función que se pasa a otra función como argumento y se ejecuta después de que una operación específica se completa. En JavaScript, esto es esencial para manejar operaciones asíncronas, como solicitudes de red o temporizadores, ya que permite que el programa continúe ejecutándose sin bloqueos mientras espera que se complete la operación.

```
function mostrarMensaje() {
    console.log("Hola, este mensaje se muestra después de 3 segundos");
}

// setTimeout usa un callback para ejecutar mostrarMensaje después de 3000
milisegundos
setTimeout(mostrarMensaje, 3000);
```

Callbacks

Veamos otro ejemplo:

La función **procesarElemento** toma un elemento y un callback como argumentos.

Usa **setTimeout** para simular una operación asíncrona que tarda 1 segundo.

Después del retraso, llama al callback con un mensaje que incluye el elemento procesado.

Luego de la definición de elementos, usamos **forEach** para iterar sobre cada elemento en el arreglo elementos.

Para cada elemento, invocamos a **procesarElemento**, pasando el elemento y un callback que imprime el resultado.

```
function procesarElemento(elemento, callback) {
    setTimeout(() => {
        callback("Elemento procesado: " + elemento);
    }, 1000);
}

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

elementos.forEach(elemento => {
    procesarElemento(elemento, resultado => {
        console.log(resultado);
    });
});
```

Este ejemplo demuestra cómo los callbacks pueden ser utilizados para manejar la asincronía en operaciones que implican iterar sobre un conjunto de datos y procesar cada ítem de manera asíncrona. Esto es común en aplicaciones web donde, por ejemplo, podrías necesitar cargar o procesar una lista de recursos obtenidos desde un servidor.

Callback Hell

El Callback Hell, o "Pyramid of Doom", es una situación donde múltiples callbacks se anidan unos dentro de otros. Esto puede ocurrir cuando varias operaciones asíncronas dependen unas de otras. Esto hace que el código sea difícil de leer y mantener.

Problemas Asociados

Legibilidad Reducida: La profundidad de anidamiento hace que el código sea difícil de seguir.

Gestión de Errores Complicada: Cada nivel de anidación requiere su propio manejo de errores, lo que puede llevar a código repetitivo y complicado.

Mantenimiento Difícil: Modificar el código en uno de los niveles de anidación puede afectar a los niveles subsecuentes.

Callback Hell

El Callback Hell, o "Pyramid of Doom", es una situación donde múltiples callbacks se anidan unos dentro de otros. Esto puede ocurrir cuando varias operaciones asíncronas dependen unas de otras. Esto hace que el código sea difícil de leer y mantener.

Problemas Asociados:

Legibilidad Reducida: La profundidad de anidamiento hace que el código sea difícil de seguir.

Gestión de Errores Complicada: Cada nivel de anidación requiere su propio manejo de errores, lo que puede llevar a código repetitivo y complicado.

Mantenimiento Difícil: Modificar el código en uno de los niveles de anidación puede afectar a los niveles subsecuentes.

Callback Hell

Ejemplo de callback Hell

Inicio del Proceso: Se imprime un mensaje inicial para indicar el comienzo.

Primera Tarea: Se utiliza setTimeout para simular una tarea que tarda 1 segundo. Una vez completada, imprime un mensaje.

Segunda Tarea: Dentro del primer setTimeout, se inicia otro setTimeout después de que la primera tarea ha completado. Esto simula otra tarea que también tarda 1 segundo.

Tercera Tarea: De manera similar, dentro del segundo setTimeout, se inicia un tercer setTimeout.

Después de completar esta tarea, imprime un mensaje final.

```
console.log("Inicio del proceso");

setTimeout(() => {
    console.log("Primera tarea completada");
    setTimeout(() => {
        console.log("Segunda tarea completada");
        setTimeout(() => {
            console.log("Tercera tarea completada");
            console.log("Todos los procesos terminados");
        }, 1000); // Tercera tarea tarda 1 segundo
    }, 1000); // Segunda tarea tarda 1 segundo
}, 1000); // Primera tarea tarda 1 segundo
```

Este código ilustra un **"Callback Hell"** porque las funciones de callback están anidadas dentro de otras funciones de callback, creando un código que se desplaza hacia la derecha y es difícil de leer y mantener. Esta estructura puede llevar a errores y dificultades al intentar rastrear el flujo de ejecución del programa, especialmente en aplicaciones más complejas donde las operaciones asíncronas son comunes.

Callback Hell (Solución)

Función tareaAsincrona: Esta función toma un mensaje como argumento y retorna una promesa.

La promesa se resuelve después de un retraso de 1 segundo, durante el cual se imprime el mensaje.

Encadenamiento de Promesas: Iniciamos el proceso con un mensaje. Llamamos a **tareaAsincrona** para la primera tarea y esperamos a que se resuelva antes de llamar a la siguiente tarea.

Utilizamos **.then()** para encadenar cada tarea secuencialmente, lo que asegura que cada tarea comience sólo después de que la anterior haya terminado. Al final, imprimimos un mensaje cuando todas las tareas han terminado.

Manejo de Errores: Usamos **.catch()** al final del encadenamiento para capturar y manejar cualquier error que pueda ocurrir durante las operaciones asíncronas.

```
function tareaAsincrona(mensaje) {
    return new Promise(resolve => {
        setTimeout(() => {
            console.log(mensaje);
            resolve();
        }, 1000);
    });
}

console.log("Inicio del proceso");

tareaAsincrona("Primera tarea completada")
    .then(() => tareaAsincrona("Segunda tarea completada"))
    .then(() => tareaAsincrona("Tercera tarea completada"))
    .then(() => {
        console.log("Todos los procesos terminados");
    })
    .catch(error => {
        console.error("Error durante el proceso", error);
    });
```

Callback Hell

```
console.log("Inicio del proceso");

setTimeout(() => {
    console.log("Primera tarea completada");
    setTimeout(() => {
        console.log("Segunda tarea completada");
        setTimeout(() => {
            console.log("Tercera tarea completada");
            console.log("Todos los procesos
terminados");
        }, 1000); // Tercera tarea tarda 1 segundo
    }, 1000); // Segunda tarea tarda 1 segundo
}, 1000); // Primera tarea tarda 1 segundo
```

Promesas

```
function tareaAsincrona(mensaje) {
    return new Promise(resolve => {
        setTimeout(() => {
            console.log(mensaje);
            resolve();
        }, 1000);
    });
}

console.log("Inicio del proceso");

tareaAsincrona("Primera tarea completada")
    .then(() => tareaAsincrona("Segunda tarea completada"))
    .then(() => tareaAsincrona("Tercera tarea completada"))
    .then(() => {
        console.log("Todos los procesos terminados");
    })
    .catch(error => {
        console.error("Error durante el proceso", error);
    });
```

Comparando ambos enfoques, las promesas no solo evitan el anidamiento profundo y mejora la legibilidad del código, sino que también facilita el manejo de errores y el flujo de control en general.

Material extra

Artículos de interés

Material de lectura:

- Async function: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/async_function*
- await: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await

Videos:

- Callbacks, Promesas y Async Await en Javascript: https://www.youtube.com/watch?v=a91rU4Jk1KU
- Callbacks VS Promises en JavaScript. ¡Entiende las diferencias y la importancia de cada una!: https://www.youtube.com/watch?v=frm0CHyeSbE

No te olvides de dar el presente

Recordá:

- Revisar la Cartelera de Novedades.
- Hacer tus consultas en el Foro.
- Realizar los Ejercicios obligatorios.

Todo en el Aula Virtual.

Muchas gracias por tu atención.

Nos vemos pronto