3.2. Programación asíncrona y promesas

LENGUAJES DE PROGRAMACIÓN NIVEL 5. UNIDAD 3

JAVASCRIPT - DESAROLLO FRONTEND AVANZADO

Financiado por la Unión Europea

Plan de Recuperación, Transformación y Resiliencia

Contenido

Introducción a la programación asíncrona:

La programación asíncrona es un paradigma de programación que permite que ciertas operaciones se ejecuten de manera independiente y sin bloquear la ejecución del resto del código. En lugar de esperar a que una operación se complete antes de continuar con la siguiente, la programación asíncrona permite que el programa continúe ejecutando otras tareas mientras espera que una operación específica termine en segundo plano.

Importancia de la programación asíncrona en el desarrollo web moderno:

La programación asíncrona es fundamental en el desarrollo web moderno debido a la naturaleza del entorno de ejecución del navegador. En las aplicaciones web, es común que se realicen operaciones que pueden llevar tiempo, como solicitudes a servidores remotos, procesamiento de datos o manipulación del DOM. Utilizar programación asíncrona permite que estas operaciones se realicen de manera eficiente sin bloquear la interfaz de usuario, lo que mejora la experiencia del usuario al evitar retrasos perceptibles.

Además, en un entorno web en el que múltiples usuarios pueden interactuar simultáneamente con una aplicación, la programación asíncrona es esencial para manejar de manera efectiva las solicitudes concurrentes y garantizar una experiencia fluida para todos los usuarios.

Ejemplos de situaciones en las que se necesita programación asíncrona:

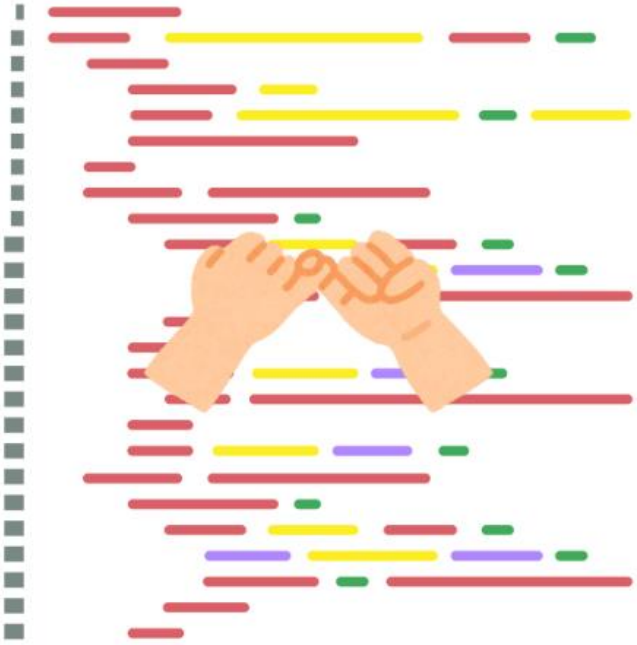
- **Solicitudes HTTP:**
  Cuando se realiza una solicitud a un servidor web para recuperar datos, como al cargar una página web o al enviar información a través de un formulario, es fundamental que esta operación se realice de manera asíncrona para evitar bloquear la interfaz de usuario.
- **Operaciones de E/S (Entrada/Salida):**
  La lectura y escritura de archivos, el acceso a bases de datos y otras operaciones de entrada/salida pueden ser procesos lentos que deben manejarse de forma asíncrona para no afectar la capacidad de respuesta de la aplicación.
- **Temporizadores y eventos del usuario:**
  El manejo de eventos del usuario, como clics de ratón o pulsaciones de teclas, así como el uso de temporizadores para realizar acciones periódicas, son ejemplos comunes de situaciones en las que se necesita programación asíncrona para responder de manera rápida y eficiente a las interacciones del usuario.

Conceptos básicos de asincronía en JavaScript:

La asincronía en JavaScript es un concepto fundamental que permite que ciertas operaciones se ejecuten de manera independiente y no bloqueante. Aquí tienes una descripción de los conceptos básicos:

- **Modelo de ejecución asíncrona en JavaScript:**
  En JavaScript, las operaciones asíncronas se ejecutan en paralelo al resto del código, lo que significa que no bloquean la ejecución de otras tareas.
  Esto se logra utilizando mecanismos como callbacks, promesas y async/await.
- **Diferencias entre Código Síncrono y Asíncrono:**
  - En el código síncrono, las instrucciones se ejecutan secuencialmente, una después de la otra, y el programa espera a que una operación termine antes de pasar a la siguiente.
  - En el código asíncrono, las operaciones se ejecutan en segundo plano y el programa continúa ejecutando otras tareas mientras espera que estas operaciones asíncronas se completen.
- **Introducción a Callbacks:**
  - Los callbacks son funciones que se pasan como argumentos a otras funciones y se ejecutan después de que una operación asíncrona haya finalizado. Son un mecanismo fundamental para manejar la asincronía en JavaScript.
  - Los callbacks permiten definir qué hacer después de que una operación asíncrona haya concluido, lo que permite realizar acciones como procesar datos, actualizar la interfaz de usuario o manejar errores.

Introducción a las promesas:



Las promesas son un concepto fundamental en la programación asíncrona en JavaScript, proporcionando una forma más estructurada y flexible de manejar operaciones asíncronas. Aquí tienes una descripción de los aspectos clave:

Definición y papel en la programación asíncrona:

Una promesa representa un valor que puede estar disponible ahora, en el futuro, o nunca.

Su principal función es gestionar el resultado eventual de una operación asíncrona, ya sea una tarea que se completa satisfactoriamente o una que falla.

Ventajas de utilizar promesas:

- Las promesas ofrecen una sintaxis más clara y legible que los callbacks anidados, lo que facilita la comprensión del flujo de control del programa.
- Permiten encadenar operaciones asíncronas de forma secuencial, evitando el problema conocido como "callback hell" o "infierno de callbacks".

- Proporcionan un mecanismo integrado para manejar errores, lo que simplifica la gestión de excepciones en código asíncrono.

Sintaxis básica:

Para crear una promesa, se utiliza el constructor Promise, que toma una función con dos parámetros: resolve y reject.

- **resolve** se utiliza para indicar que la operación asíncrona se ha completado con éxito y para pasar el resultado deseado.
- **reject** se utiliza para indicar que la operación ha fallado y para pasar un error asociado.

Lógica de las promesas

las promesas se crean y consumen utilizando cualquiera de las dos opciones que se han planteado al crearlas.

- **Creación de promesas:** Las promesas se crean utilizando el constructor Promise, donde se especifica una función que toma dos argumentos: resolve y reject. Dentro de esta función, se realiza una operación asíncrona y, una vez completada, se llama a resolve con el resultado deseado si la operación fue exitosa, o a reject con un error si la operación falló.
- **Consumo de promesas:** Una vez que se crea una promesa, se puede consumir utilizando el método then para manejar el resultado exitoso y el método catch para manejar cualquier error que ocurra durante la ejecución de la promesa. También se pueden encadenar múltiples métodos then para realizar operaciones adicionales en el resultado de la promesa.

Ambos aspectos son esenciales para trabajar de manera efectiva con promesas en JavaScript: crearlas adecuadamente para representar operaciones asincrónicas y consumirlas de manera apropiada para manejar sus resultados y posibles errores.

Ejemplo:

1. Ejemplo de creación de una promesa:

```
const miPromesa = new Promise((resolve, reject) => {
    // Operación asíncrona
    if (operacionExitosa) {
        resolve(resultado);
    } else {
        reject(error);
    }
});
```

[1]

Este código crea una promesa en JavaScript. Una promesa es básicamente una manera de decirle a JavaScript que haga algo ahora y luego hacer algo más cuando esté listo.

En este caso específico, la promesa se crea utilizando la sintaxis new Promise, que toma una función como argumento. Dentro de esta función, realizamos alguna operación asíncrona, que podría ser una solicitud de red, una operación de lectura de archivos, o cualquier otra tarea que lleve tiempo.

Si la operación asíncrona se realiza con éxito, llamamos a la función resolve y le pasamos el resultado deseado. Si algo sale mal durante la operación, llamamos a la función reject y le pasamos un error que describa lo que sucedió.

En resumen, esta promesa representa la realización de una tarea asíncrona, con la garantía de que se realizará alguna acción (ya sea

[1] Accesibilidad

```
const miPromesa = new Promise( (resolve, reject) => {
        // Operación asincrona
        if (operacionExitosa) {
                resolve(resultado);
        } else {
                reject(error);
        }
});
```

manejar un resultado exitoso o un error) una vez que se complete la tarea.

2. Para consumir una promesa, se utiliza el método then para manejar el caso de éxito y catch para manejar errores:

```
miPromesa.then(resultado => {
    // Manejar resultado exitoso
}).catch(error => {
    // Manejar error
});
```

[2]

Este código en JavaScript está utilizando el método then y catch para manejar el resultado de una promesa llamada miPromesa.

a. El método then se ejecutará cuando la promesa se resuelva con éxito. En este caso, estamos pasando una función que tomará el resultado de la promesa (lo que se pasó a la función resolve en la promesa original) y realizará alguna operación con él. En otras palabras, esta función manejará el resultado exitoso de la promesa.
b. El método catch se ejecutará si la promesa es rechazada, es decir, si ocurre algún error durante la ejecución de la promesa. Aquí estamos pasando una función que tomará el error que se pasó a la función reject en la promesa original y realizará alguna acción para manejar ese error.

En resumen, este código establece cómo manejar tanto el éxito como el error de la promesa miPromesa, proporcionando funciones para ejecutar en cada caso. Esto permite controlar el flujo de la aplicación y manejar diferentes escenarios según el resultado de la operación asíncrona.

[2] Accesibilidad

```
miPromesa.then (resultado = {
                    //Manejar resultado exitoso
        }).catch(error => {
                    // Manejar error
});
```

Las promesas son una herramienta poderosa para trabajar con operaciones asíncronas en JavaScript, proporcionando una forma más clara y estructurada de manejar el flujo de control y los errores. Su uso se ha vuelto estándar en el desarrollo moderno de aplicaciones web debido a su facilidad de uso y robustez.

Manejo de promesas:

El manejo de promesas en JavaScript es fundamental para gestionar operaciones asíncronas de manera efectiva. Los métodos then() y catch() son esenciales para trabajar con el resultado y los errores de las promesas.

- then() y catch(): El método then() se utiliza para manejar el resultado exitoso de una promesa. Permite encadenar operaciones adicionales que se ejecutarán después de que la promesa se resuelva satisfactoriamente. Por otro lado, el método catch() se utiliza para manejar cualquier error que pueda ocurrir durante la ejecución de la promesa.

```
miPromesa.then(resultado => {
    // Manejar resultado exitoso
}).catch(error => {
    // Manejar error
});
```

3

- Encadenamiento de promesas: Una de las características más poderosas de las promesas es su capacidad para encadenarse, lo que permite realizar operaciones asíncronas en secuencia.

[3] Accesibilidad

```
miPromesa.then(resultado => {
        // Manejar resultado exitoso
}).catch(error => {
        // Manejar error
});
```

Esto se logra devolviendo otra promesa dentro del método then(), lo que permite que la salida de una promesa se convierta en la entrada de la siguiente.

```
miPrimeraPromesa.then(resultado => {
    // Realizar operaciones con el resultado
    return miSegundaPromesa; // Devolver otra promesa
}).then(nuevoResultado => {
    // Manejar el nuevo resultado
}).catch(error => {
    // Manejar errores en cualquiera de las promesas
});
```

4

Este código realiza operaciones de forma asíncrona en secuencia. Primero espera a que miPrimeraPromesa se resuelva, luego realiza operaciones con su resultado y, finalmente, espera a que miSegundaPromesa se resuelva y maneja su resultado. Si hay algún error en el proceso, se captura y se maneja adecuadamente.

- Casos de uso: Las promesas se utilizan comúnmente para manejar operaciones asíncronas en el desarrollo web, como solicitudes HTTP, operaciones de lectura/escritura de archivos y otras tareas que requieren tiempo.

[4] Accesibilidad

```
miPrimeraPromesa. then (resultado => {
        // Realizar operaciones con el resultado
        return miSegundaPromesa; // Devolver otra promesa
}).then(nuevoResultado => {
        // Manejar el nuevo resultado
}).catch(error => {
        // Manejar errores en cualquiera de las promesas
});
```

Async/Await:

El uso de las palabras clave async y await en JavaScript proporciona una forma más clara y legible de trabajar con promesas y operaciones asíncronas.

Definición

- **async:** Esta palabra clave se utiliza antes de una función para indicar que la función devuelve una promesa. Permite utilizar la palabra clave await dentro de ella para esperar la resolución de otras promesas sin bloquear la ejecución del código.
- **await:** Esta palabra clave se utiliza dentro de una función async para indicar que se debe esperar a que una promesa se resuelva antes de continuar con la ejecución del código.

Lógica de uso

Imagina que estás en una fila esperando tu turno para comprar un helado. Tienes un boleto que te garantiza que obtendrás tu helado cuando llegue tu turno. Este boleto es como una promesa en JavaScript.

Ahora, async y await son como pedirle a alguien que espere en la fila por ti:

- **con async,** le dices a JavaScript que una función puede contener promesas y que debe esperar a que se resuelvan antes de continuar ejecutando el código.
- **con await,** le dices a JavaScript que espere a que se resuelva una promesa antes de continuar ejecutando la función.

Entonces, imagina que estás en una función y tienes que esperar a que se resuelvan varias promesas para continuar. Usas async para indicar que tu función contiene promesas y luego usas await para esperar a que cada promesa se resuelva antes de continuar con el código. Es como si delegaras la espera en alguien más para que puedas seguir haciendo otras cosas mientras tanto.

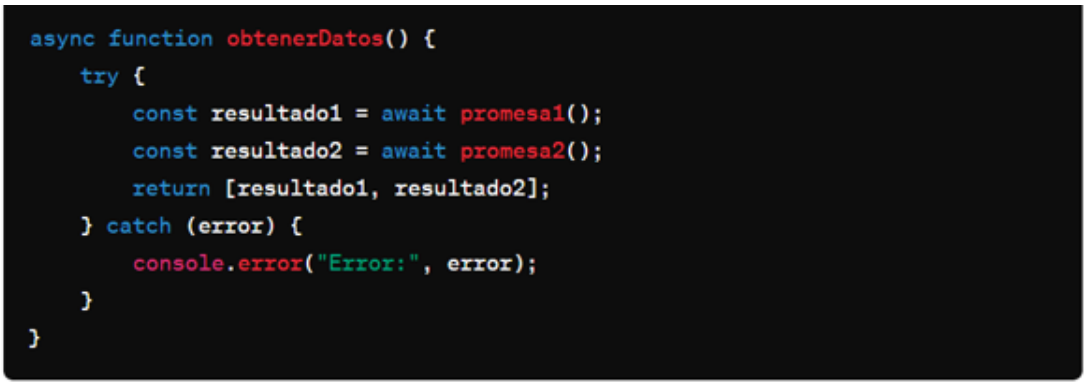
Casos de uso

Puedes necesitar usar async y await en situaciones donde tengas que realizar múltiples operaciones asíncronas y desees escribir un código más claro y legible. Aquí hay algunas situaciones comunes:

- Cuando necesitas hacer varias llamadas a API y deseas asegurarte de que todas se completen antes de continuar con el siguiente paso en tu programa.
- Cuando necesitas realizar operaciones de E/S (entrada/salida) en el navegador, como leer o escribir archivos, y quieres esperar a que se completen antes de continuar.
- Cuando estás trabajando con bases de datos y necesitas esperar a que se realicen consultas antes de proceder con otras operaciones.
- Cuando necesitas manejar errores de manera más efectiva en tu código asíncrono, especialmente cuando se encadenan múltiples promesas.

En resumen, async y await son útiles en cualquier situación donde necesites manejar operaciones asíncronas de manera más clara y concisa en tu código JavaScript.

Ejemplo de uso



```
async function obtenerDatos() {
    try {
        const resultado1 = await promesa1();
        const resultado2 = await promesa2();
        return [resultado1, resultado2];
    } catch (error) {
        console.error("Error:", error);
    }
}
```

5

En este ejemplo, la función obtenerDatos utiliza async para indicar que es asíncrona y utiliza await para esperar a que promesa1 y promesa2 se resuelvan antes de continuar. Esto simplifica la estructura del código y hace que sea más fácil de entender.

Ventajas de uso:

a. **Legibilidad:** Permite escribir código asíncrono de manera más clara y secuencial, sin el uso excesivo de callbacks o encadenamiento de .then().
b. **Manejo de errores:** Facilita el manejo de errores utilizando bloques try/catch, lo que mejora la detección y gestión de fallos en operaciones asíncronas.

[5] Accesibilidad

```
async function obtenerDatos() {
    try {
        const resultado1 = await promesa1();
        const resultado2 = await promesa2();
        return [resultado1, resultado2];
    } catch (error) {
        console.error("Error”, error);
    }
}
```

Patrones y mejores prácticas:

En el desarrollo de código asíncrono, es importante seguir algunos patrones y mejores prácticas para garantizar que nuestro código sea limpio, mantenible y eficiente.

Aquí encontrarás algunas pautas importantes:

- **Utilizar nombres descriptivos:** Asegúrate de dar nombres descriptivos a tus funciones asíncronas y variables para que otros desarrolladores (y tu futuro yo) puedan entender fácilmente qué hace cada parte del código.
- **Evitar anidar demasiadas promesas:** El anidamiento excesivo de promesas puede conducir a lo que se conoce como "promesas del infierno" o "callback hell". En su lugar, utiliza métodos como then, catch y finally, así como async/await para mantener un código más limpio y legible.
- **Manejar errores adecuadamente:** Siempre asegúrate de manejar los errores correctamente en tu código asíncrono. Usa bloques try-catch alrededor de tus llamadas asíncronas para capturar errores y manejarlos de manera adecuada.
- **Evitar promesas no manejadas:** Siempre debes manejar las promesas rechazadas para evitar errores no detectados en tu aplicación. Puedes usar el método catch al final de una cadena de promesas para capturar cualquier error que ocurra en cualquier parte de la cadena.

- **Considerar el rendimiento:** Ten en cuenta el rendimiento al trabajar con operaciones asíncronas. Por ejemplo, evita realizar demasiadas solicitudes de red simultáneas, ya que esto puede afectar el rendimiento de tu aplicación. En su lugar, considera utilizar técnicas como la agrupación de solicitudes o el uso de cachés para reducir la carga en el servidor.
- **Separar la lógica de la interfaz de usuario:** Cuando sea posible, separa la lógica de la interfaz de usuario de las operaciones asíncronas. Esto hace que tu código sea más modular y fácil de mantener, ya que puedes reutilizar las funciones asíncronas en diferentes partes de tu aplicación.

Siguiendo estas mejores prácticas y consideraciones, puedes escribir código asíncrono más limpio, mantenible y eficiente en tus proyectos de JavaScript.

Conclusiones:

En este módulo, hemos explorado los conceptos fundamentales de la programación asíncrona y el manejo de promesas en JavaScript. Aquí hay una recapitulación de los puntos clave aprendidos:

- Aprendimos qué es la programación asíncrona y por qué es importante en el desarrollo web moderno. La asincronía nos permite realizar operaciones sin bloquear la ejecución del programa, lo que mejora la experiencia del usuario al evitar la espera innecesaria.
- Exploramos los conceptos básicos de las promesas, que son objetos que representan un valor que puede estar disponible ahora, en el futuro o nunca. Las promesas nos permiten trabajar de manera más efectiva con operaciones asíncronas y manejar los resultados exitosos o los errores de manera más clara y legible.
- Descubrimos cómo utilizar async/await, una característica introducida en ECMAScript 2017, que simplifica aún más el manejo de promesas al permitirnos escribir código asíncrono de manera síncrona, lo que hace que nuestro código sea más fácil de leer y entender.

Es crucial dominar estos conceptos ya que son fundamentales en el desarrollo frontend avanzado de JavaScript. Muchas aplicaciones web modernas dependen en gran medida de la asincronía para realizar operaciones como solicitudes de red, manipulación de datos y actualizaciones de la interfaz de usuario de manera eficiente.

Por lo tanto, te animo a practicar y experimentar con estos conceptos en sus propios proyectos. Cuanto más te familiarices con la programación asíncrona y el manejo de promesas, más capaz serás de construir aplicaciones web robustas y escalables. ***¡No tengas miedo de sumergirte en la asincronía y descubrir todo lo que puedes lograr con ella!***

Despedida y recursos adicionales:

¡Enhorabuena por completar este módulo sobre programación asíncrona y promesas en JavaScript! Espero que hayas encontrado el contenido útil y que te sientas más cómodo trabajando con operaciones asíncronas en tus proyectos.

Recuerda que la programación asíncrona es una habilidad fundamental en el desarrollo web moderno, y dominarla te abrirá muchas puertas para crear aplicaciones web más rápidas, interactivas y eficientes.

Para seguir aprendiendo y profundizando en estos temas, te recomiendo explorar los siguientes recursos adicionales:

- Documentación oficial de JavaScript:
  La documentación en línea de JavaScript, especialmente la sección dedicada a promesas y async/await en el sitio web de Mozilla Developer Network (MDN), es una excelente fuente de información detallada y ejemplos prácticos.
- Comunidades en línea: Únete a comunidades en línea como Stack Overflow, Reddit (r/javascript), y Discord (por ejemplo, en servidores como el de freeCodeCamp) donde puedes hacer preguntas, participar en discusiones y aprender de otros desarrolladores.
- Proyectos de código abierto: Explora proyectos de código abierto en plataformas como GitHub y GitLab que utilicen promesas y async/await en su código. Analizar el código de proyectos reales te ayudará a entender cómo se aplican estos conceptos en situaciones prácticas.
- Práctica personal: No hay mejor manera de aprender que practicar por ti mismo. Crea tus propios proyectos y desafíos donde puedas aplicar lo que has aprendido sobre programación asíncrona y promesas. Experimenta con diferentes escenarios y problemas para mejorar tus habilidades.

Recuerda que la práctica constante es la clave para mejorar en programación. ***¡No tengas miedo de experimentar y poner en práctica lo que has aprendido!***

¡Sigue adelante y sigue aprendiendo! Estoy seguro de que, con dedicación y esfuerzo, llegarás lejos en tu viaje como desarrollador web. ***¡Mucho éxito en tus futuros proyectos!***