«Talento Tech»

Back-End

Node JS

Clase 06

Buenos Aires Ciudad

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Clase N° 6 - Asincronismo

Temario:

1. Fundamentos del asincronismo
2. Manejo de promesas:
    - callbacks
    - then, catch & finally
    - async & await
3. Fetch: consumiendo datos externos

Objetivos de la Clase

En esta clase, los estudiantes comprenderán qué es el asincronismo y su importancia en la programación moderna. Aprenderán a manejar tareas asíncronas utilizando promesas con los métodos **then**, **catch** y **finally**, y aplicarán las palabras clave **async** y **await** para simplificar y estructurar el código basado en promesas. Además, explorarán cómo consumir datos externos mediante la API Fetch utilizando peticiones **GET** y **POST**, al mismo tiempo que aprenderán a manejar errores y procesar respuestas al interactuar con APIs externas.

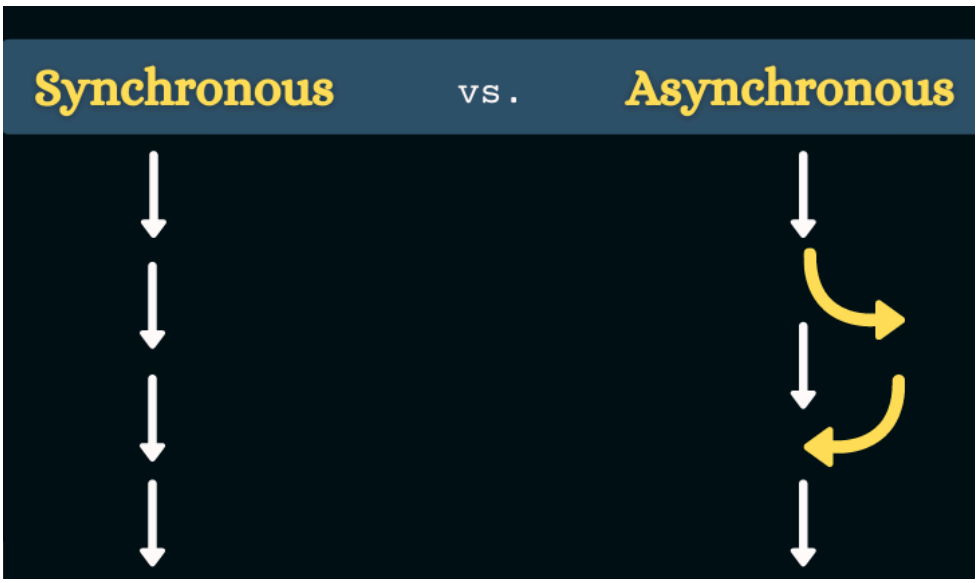
Buenos Aires aprende
Agencia de Habilidades para el Futuro

Fundamentos del asincronismo

Definición:

El asincronismo es un paradigma de programación que permite que una tarea pueda ejecutarse en segundo plano sin bloquear la ejecución del programa principal.

En otras palabras, se pueden realizar varias tareas simultáneamente, sin necesidad de esperar a que una tarea se complete antes de comenzar la siguiente. El asincronismo es importante porque nos permite construir aplicaciones más eficientes y más rápidas.



¿Cómo funciona en Javascript?

JavaScript del lado del cliente es un lenguaje de programación basado en eventos, lo que significa que el código se ejecuta en respuesta a eventos que ocurren en el entorno. El asincronismo en JavaScript se logra mediante la utilización de callbacks, promesas, y async/await.

Es importante manejar el asincronismo en JavaScript para permitir que las tareas que tardan mucho tiempo en ejecutarse se realicen en segundo plano sin bloquear la ejecución del programa principal. Si no se maneja el asincronismo adecuadamente, se corre el riesgo de que el programa falle o incluso se bloquee por completo, lo que puede afectar negativamente la experiencia del usuario.

Como vimos en clases anteriores, JavaScript es un lenguaje de programación de un solo subproceso (single-threaded), lo que significa que todas las tareas deben ejecutarse en una sola secuencia de comandos. A diferencia de otros lenguajes de programación como Java o C#, que pueden ejecutar múltiples subprocesos simultáneamente, JavaScript solo puede ejecutar una tarea a la vez.

Sin embargo, aunque JavaScript es de un solo subproceso, puede manejar múltiples tareas asincrónicas simultáneamente mediante el uso de mecanismos como el Event Loop y las promesas. El Event Loop se encarga de manejar la cola de tareas, lo que permite que las tareas asincrónicas se ejecuten en segundo plano mientras se sigue ejecutando el programa principal.

¿Cómo funciona el Single Thread?

JavaScript maneja el Single Thread utilizando una estructura de datos llamada Call Stack y técnicas de asincronismo como Callbacks, Promises y Async/Await, junto con el mecanismo de Event Loop, para permitir que el programa realice múltiples tareas en segundo plano sin bloquear la ejecución del programa principal.

Cuando una función se llama, se agrega a la Call Stack y se ejecuta de forma sincrónica, lo que significa que el programa debe esperar hasta que la función se complete antes de continuar con la siguiente tarea. Si la función tarda mucho tiempo en ejecutarse, puede hacer que el programa se bloquee.

Por eso muchas veces necesitamos crear procesos mediante una función asincrónica que espere el resultado mientras el programa se sigue ejecutando.

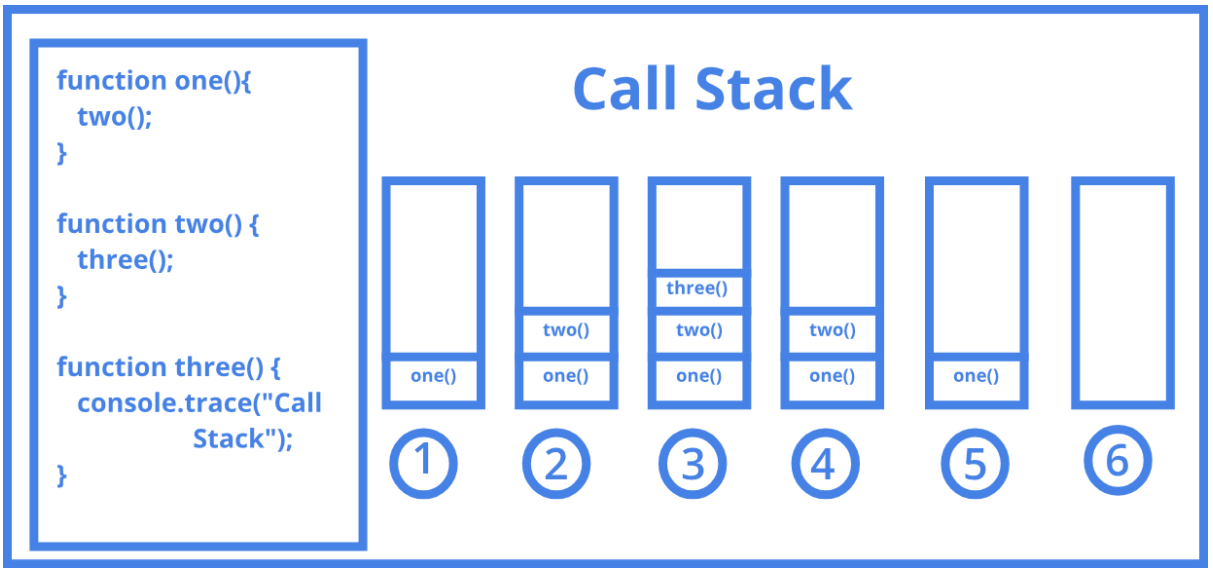
Esta función se coloca en una cola de tareas llamada Callback Queue, que espera hasta que todas las tareas sincrónicas se completen y la Call Stack esté vacía antes de ejecutar

dicha función. Esto se logra mediante el uso de un mecanismo llamado Event Loop, que se encarga de monitorear la Call Stack y la Callback Queue y ejecutar la siguiente tarea en la cola de tareas sólo cuando la Call Stack está vacía

Call Stack

Para entender cómo funciona el asincronismo en JavaScript, es necesario entender algunos conceptos clave. El primero de ellos es **Call Stack** (pila de llamadas), que es una estructura de datos que mantiene un registro de las funciones que están siendo ejecutadas en un momento dado.

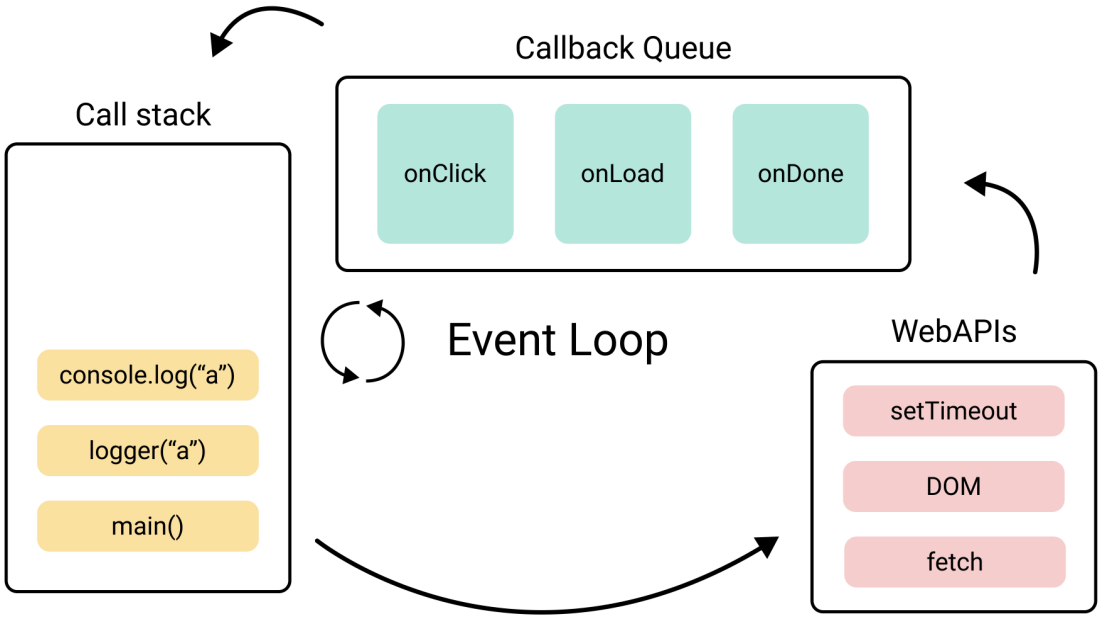
Cuando una función se llama a sí misma, se agrega a la pila de llamadas. Cuando se completa la ejecución de una función, se elimina de la pila de llamadas. La pila de llamadas en JavaScript es importante porque es una estructura de datos que ayuda a controlar el flujo de la ejecución del programa.



Event Loop

El segundo concepto clave es el **Event Loop** (bucle de eventos), que es un mecanismo que permite que el código asíncrono se ejecute en segundo plano sin bloquear la ejecución del programa principal. El Event Loop en JavaScript es una estructura que mantiene un registro de los eventos que ocurren en el entorno.

Cuando ocurre un evento, el Event Loop se encarga de manejar ese evento y agregarlo a una cola de callbacks o Callback Queue. La cola de callbacks es una estructura de datos que mantiene un registro de las funciones que se deben ejecutar en respuesta a un evento.



Buenos Aires aprende
Agencia de Habilidades para el Futuro

Callback Queue

La **Callback Queue** es una estructura de datos en JavaScript que se utiliza para almacenar las funciones callback (subprocesos relacionados) o resultados que se deben ejecutar o devolver después de que se hayan completado las tareas asincrónicas en segundo plano.

Cuando se llama a una función asincrónica en JavaScript, como por ejemplo una solicitud de datos a una fuente externa o una animación en el navegador, la función no se ejecuta de forma sincrónica. En su lugar, se coloca en una cola de tareas llamada Callback Queue. Una vez que se han completado todas las tareas sincrónicas y la Call Stack está vacía, el Event Loop se encarga de ejecutar las funciones en la Callback Queue en orden de llegada.

Un ejemplo común de uso de la Callback Queue es consulta a una API externa (fuente de datos). Cuando se realiza una dicha solicitud, el programa no espera a que se complete la solicitud antes de continuar ejecutando las demás tareas. En su lugar, la solicitud se realiza en segundo plano y la función de callback se agrega a la Callback Queue. Cuando la solicitud a la API se completa, la función de callback se ejecuta y se procesa la respuesta.

Manejo de Promesas

Vimos como Javascript maneja procesos asincrónicos internamente a pesar de contar un con solo hilo de ejecución o “Single Thread”. En la práctica conocer esto es determinante para saber cuando debemos crear funciones o procesos síncronos o asíncronos.

En esta ocasión aprenderemos a crear estos procesos asíncronos y para ello contamos con diversas herramientas.

Callbacks

Un callback es una función que se pasa como argumento a otra función y se ejecuta cuando se completa la tarea. El siguiente ejemplo ilustra el uso de un callback:

```
function taskAsync(callback) {
  setTimeout(function() {
    console.log('Tarea asincrónica completada.');
    callback();
  }, 3000)
};

console.log('Inicio de la tarea.');
taskAsync(function() {
  console.log('Fin de la tarea.');
});
```

En este ejemplo, la función **taskAsync** es una función asíncrona que simula una tarea que tarda 3 segundos en completarse.

La función acepta un argumento de **callback**, que se ejecuta cuando se completa la tarea.

En este caso, el **callback** simplemente muestra un mensaje en la consola indicando que la tarea ha finalizado.

```
Inicio de la tarea.
Tarea asincronica completada.
Fin de la tarea.
```

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Promesas

Las promesas son una forma más moderna y elegante de lograr el asincronismo en JavaScript. Una promesa es un objeto que representa el resultado eventual de una tarea asíncrona. El resultado eventual puede ser un valor o un error. Es por eso que las promesas cuentan con dos estados finales: **Resolved** y **Rejected**.

Cual de los dos estados tomará como resultado la función se definirá al momento de crear la **Promesa** indicando que devolver si el proceso fue exitoso o erróneo.

Por otra parte, para poder recuperar el resultado de una **Promesa** utilizaremos los métodos `.then()`, `.catch()` y `.finally()`.

Veamos el siguiente ejemplo donde se muestra cómo declarar una promesa:

```
function taskAsync() {
  return new Promise(function (resolve, reject) {
    setTimeout(function() {
    if (Math.random() < 0.5) {
      resolve('Tarea asincrónica completada.')
    } else {
      reject(new Error('Tarea asincrónica fallida.'))
    }
    }, 3000);
  });
}
```

Buenos Aires aprende
Agencia de Habilidades para el Futuro

En este ejemplo, la función **taskAsync** devuelve una promesa que se resuelve después de 3 segundos. La promesa utiliza los métodos **resolve** y **reject** que recibe automáticamente por parámetro, para indicar si la tarea se completó exitosamente o falló.

En este caso colocamos una condición **Math.random() < 5** para simular que la promesa pueda llegar a fallar, ya que de otra forma siempre sería exitosa.

Luego al momento de ejecutar esta función debemos recuperar el resultado pero eso no es posible asignándole la ejecución a una variable, ya que como el programa sigue corriendo, va a leer la línea de código y asignarle a la variable el resultado ***"Promise { <pending> }"*** dado que la ejecución de ese código asíncrono todavía ha finalizado.

```
const result = taskAsync();
console.log(result); // Promise { <pending> }
```

Es por eso que debemos utilizar los métodos anteriormente mencionados, veamos un ejemplo:

```
console.log('Inicio de la tarea.');

taskAsync()
.then((result) => console.log(result))
.catch((error) => console.log(error))
.finally(() => console.log('Fin de la tarea.'));
```

El método **then** establece que debe suceder una vez que la promesa se resuelve con éxito, mientras que el método **catch** determina qué sucederá si la promesa falla.

En ambos casos se recibe por parámetro el valor resultante al cual podemos colocarle el nombre que deseemos, como por ejemplo **result** en el `then` y **error** en el `catch`.

Por último, el método `finally` es opcional y se llama al finalizar los dos anteriores, independientemente de si la promesa se resuelve o se rechaza.

*Nota: El método **catch** también es opcional pero en caso que nuestra función falle no obtendremos un resultado válido o esperado. Por eso siempre es recomendable utilizarlo para manejar y decidir que hacer frente a posibles errores.

Async & Await

El **async/await** es una forma más moderna de lograr el asincronismo en JavaScript que se basa en las promesas.

- El `async` es una palabra clave que se utiliza para definir una función asincrónica.
- El `await` es una palabra clave que se utiliza dentro de una función asincrónica para esperar la resolución de una promesa antes de continuar con la ejecución del código.

Estas palabras claves se utilizan en reemplazo de los métodos `then`, `catch` y `finally` y siempre es recomendable utilizarlas dentro de un bloque de `try`/`catch` que nos permite ejecutar “cualquier” pieza de código dentro del `try` y capturar un fallo mediante el `catch`.

Veamos cómo quedaría el ejemplo anterior trabajando con **async/await**:

Buenos Aires aprende
Agencia de Habilidades para el Futuro

```
async function executeAsyncTask () {
  console.log('Inicio de la tarea.');
  try {
    const result = await taskAsync();
    console.log(result);
  } catch (error) {
    console.log(error);
  } finally {
    console.log('Fin de la tarea.')
  }
}

executeAsyncTask();
```

A simple vista pueden parecer similares pero el valor agregado radica en que con **async/await** podemos ejecutar múltiples funciones asíncronas una debajo de otra y esperar el resultado colocando la palabra `await`. En cambio con la forma tradicional el mismo resultado se lograría concatenando un `.then()` dentro de otro cuando los procesos son dependientes.

Conclusión:

El asincronismo es una parte importante de la programación en JavaScript. Permite que las tareas se ejecuten en segundo plano sin bloquear la ejecución del programa principal y se logra mediante el uso de mecanismos como el Event Loop y las promesas.

El async/await es una forma moderna y conveniente de lograr el asincronismo en JavaScript y se basa en las promesas. Al comprender el asincronismo y sus mecanismos subyacentes, los desarrolladores pueden escribir programas más eficientes y efectivos en JavaScript.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Fetch: consumiendo datos externos

Fetch es una API nativa de JavaScript presente tanto del lado del browser (navegador) como del servidor con Node, que se utiliza para hacer **solicitudes HTTP** a servidores web. Se considera como una alternativa a la API **XMLHttpRequest** que se utilizaba anteriormente para interactuar con servidores, es decir, un reemplazo a la manera tradicional de realizar peticiones **AJAX**.

Fetch utiliza **promesas** de JavaScript para manejar las solicitudes y las respuestas. Como vimos anteriormente las promesas son una característica de JavaScript que permiten el manejo asíncrono de código. En lugar de bloquear el hilo de ejecución mientras se espera una respuesta, las promesas permiten que otros códigos se ejecuten mientras se espera la respuesta.
Una solicitud **Fetch** se realiza utilizando la función global fetch(). Esta función devuelve una promesa que se resuelve con una respuesta **HTTP** cuando se recibe una respuesta del servidor.

Un ejemplo común de uso de Fetch es en la carga de datos de un servidor para mostrar en una página web. La siguiente es una solicitud Fetch básica que obtiene datos de un servidor y los convierte en formato JSON:

```
fetch('https://example.com/data.json')
.then((response) => response.json())
.then((data) => console.log(data));
```

El uso del método `.json()` en el primer `then` convierte la respuesta del servidor en formato **json** válido.

Otras funcionalidades importantes de Fetch:

1. **Configuración avanzada**: Puedes personalizar las solicitudes agregando un objeto de configuración. Por ejemplo, configurar los encabezados HTTP, enviar datos en el cuerpo de una solicitud POST o definir el método HTTP:
2. **Soporte para cookies y autenticación**: Usando la opción `credentials`, Fetch puede manejar cookies o enviar credenciales entre dominios.

```
const config = {
  method: 'POST',
  headers: {
      'Content-Type': 'application/json',
      'Authorization': 'Bearer token',
  },
  body: JSON.stringify({ key: 'value' }),
};

fetch('https://api.example.com/data', config)
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error('Error:', error));
```

Aunque Fetch es una herramienta nativa, existen bibliotecas de terceros como **Axios** que ofrecen una experiencia más completa y sencilla para realizar solicitudes **HTTP**.

```
import axios from 'axios';

axios.get('https://api.example.com/data')
  .then(response => console.log(response.data))
  .catch(error => console.error(error));
```

Diferencias entre Fetch y Axios:

Compatibilidad:

- Fetch es nativo y no requiere instalación, pero no está disponible en versiones antiguas de navegadores.
- Axios es una biblioteca externa y requiere instalación, pero es compatible incluso con navegadores más antiguos con un polyfill.

Manejo de respuestas:

- Fetch no lanza errores automáticamente para respuestas HTTP con códigos de error (como 404 o 500). El error debe manejarse verificando manualmente `response.ok`.
- Axios lanza errores automáticamente para respuestas fuera del rango 2xx, lo que simplifica el manejo de errores.

Facilidad de uso:

- Axios incluye muchas características útiles integradas, como tiempo de espera (`timeout`), cancelación de solicitudes y transformación automática de datos JSON en ambas direcciones.
- Fetch requiere que configures manualmente algunas de estas funcionalidades.

Soporte para JSON:

- En Fetch, debes convertir manualmente las respuestas a JSON usando `response.json()`.
- Axios convierte automáticamente las respuestas a JSON si el servidor envía datos en este formato.

Ambas herramientas son válidas, y la elección entre Fetch y Axios depende de las necesidades de tu proyecto. Axios suele ser preferido por su simplicidad y características integradas, mientras que Fetch es una solución nativa y liviana.

Ejercicio Práctico.

Ejercicio 1 - Consumir una API con Promesas

Matías y Sabrina están cada vez más interesados en tu progreso. “Es momento de adentrarte en el mundo real de las APIs”, dice Sabrina con una sonrisa desafiante.

“Queremos ver cómo manejás datos externos y cómo aprovechás las herramientas de JavaScript para hacerlo de manera eficiente”.

Matías añade: “Piensa en este ejercicio como una simulación. En proyectos reales, consumirás APIs externas todo el tiempo. Este desafío evaluará tu habilidad para hacerlo de manera estructurada y profesional”.

Misión:

1. Utiliza la API pública de Rick and Morty (docs) para obtener la lista de personajes.
2. Con las herramientas `then`, `catch` y `finally`, procesa la respuesta y devuelve por consola un **array con los primeros 5 resultados** de los 20 personajes recibidos.

Ejercicio 2 - Explorando la API con Async/Await

Impresionados por tu desempeño con promesas, Matías da un paso al frente. “El enfoque con promesas es sólido, pero en muchos casos queremos trabajar de manera más legible y fluida. Aquí es donde entra async/await. Veamos si podés replicar tu solución anterior usando esta técnica”.

Misión 2:

1. Realiza el mismo ejercicio anterior, pero esta vez usa una **función asíncrona** con `async` y `await` para consumir la API.
2. Asegúrate de manejar errores correctamente con un bloque `try/catch`.

Matías concluye: “Queremos ver un código limpio, fácil de entender y bien estructurado. Si podés manejar ambas técnicas, será una señal de que estás preparado para enfrentar tareas reales en TechLab”.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Materiales y Recursos Adicionales:

Event Loop en Node JS - Documentación Oficial

Aprende más sobre la API Fetch y sus capacidades: MDN Web Docs - Fetch API

Una alternativa a Fetch para solicitudes HTTP avanzadas: Axios GitHub Repository

---

Preguntas para Reflexionar:

- ¿Cuáles son las ventajas del manejo de asincronismo en JavaScript frente a un enfoque sincrónico?
- ¿Qué diferencias encuentras entre usar `then/catch` y `async/await`? ¿Cuándo preferirías uno sobre el otro?
- ¿Cómo puedes garantizar que tu aplicación maneje errores de manera eficiente al consumir datos externos?

Próximos Pasos:

Servidores Web: Aprenderemos sobre la comunicación web y cómo funcionan los servidores que dan vida a internet.

Patrones de Arquitectura: Conoceremos sobre los cimientos de los proyectos de programación.

Creando un Servidor Web: Daremos nuestros primeros pasos en la creación de servidores web con Node JS.

Buenos Aires Ciudad