



Agencia de Habilidades
para el Futuro

<Talento
Tech/>

Node.JS

Clase 6: Asincronismo en JavaScript

¡Les damos la bienvenida!

 Vamos a comenzar a grabar la clase.

Índice

A large yellow arrow pointing to the right, containing the number 1.

1

A large teal arrow pointing to the right, containing the number 2.

2

Asincronismo

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

Servidores Web

- ¿Cómo funciona Internet?
 - Protocolo TCP/IP
- Modelo Cliente/Servidor
- Protocolo de comunicación
- HTTP en profundidad
 - Métodos HTTP
 - Headers
 - Body
 - Códigos de estado
 - URI: URL + URN

Objetivos de la Clase

- 1** Comprender qué es el asincronismo y su importancia en la programación moderna.
- 2** Utilizar promesas con los métodos then, catch y finally para manejar tareas asíncronas.
- 3** Aplicar async y await para simplificar y estructurar el manejo de promesas.
- 4** Consumir datos externos utilizando la API Fetch con peticiones GET y POST.
- 5** Manejar errores y procesar respuestas al interactuar con APIs externas.

Fundamentos del Asincronismo

Fundamentos del Asincronismo

Es un paradigma que permite ejecutar tareas en segundo plano sin bloquear el programa principal.

Importancia

Construye aplicaciones más eficientes y rápidas, mejorando la experiencia del usuario.

JavaScript

Lenguaje basado en eventos que utiliza callbacks, promesas y async/await.



Single Thread en JavaScript

1 Ejecución Secuencial

JavaScript ejecuta tareas en una sola secuencia de comandos.

2 Event Loop

Maneja la cola de tareas para ejecutar operaciones asíncronas.

3 Call Stack

Estructura que registra las funciones en ejecución.



Call Stack y Event Loop

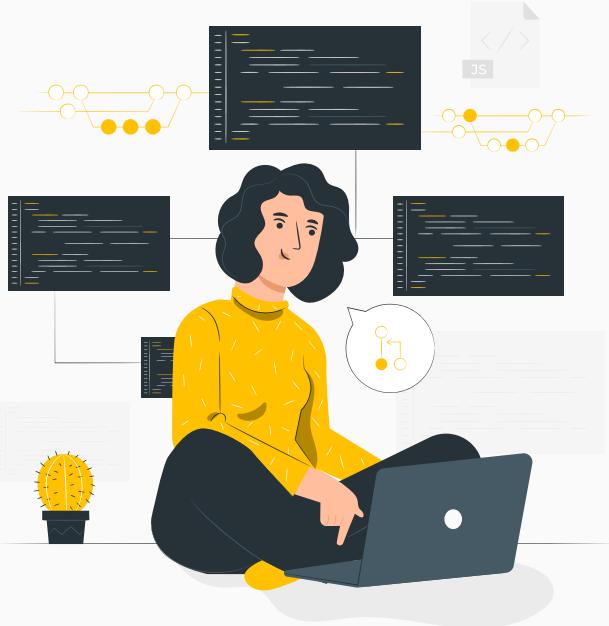
Call Stack

Registra las funciones en ejecución. Se agregan al llamarse y se eliminan al completarse.

Event Loop

Maneja eventos y agrega callbacks a la cola. Ejecuta tareas cuando el Call Stack está vacío.

Callback Queue



Almacenamiento

Guarda funciones callback y resultados de tareas asincrónicas.

Ejecución

El Event Loop ejecuta las funciones cuando el Call Stack está vacío.

Ejemplo

Consultas a APIs externas se manejan en la Callback Queue.

Manejo de promesas

Callbacks

Es una función que se pasa como argumento a otra función.

Ejecución

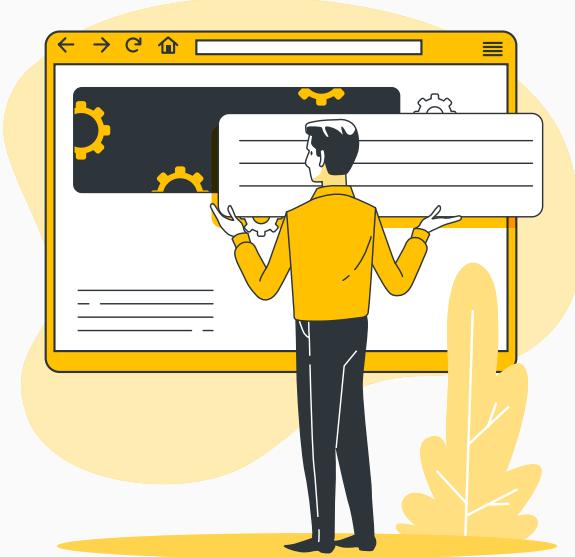
Se ejecuta cuando se completa la tarea asíncrona.

Ejemplo

taskAsync simula una tarea que tarda 3 segundos en completarse.



Objeto Promise



Es un objeto que representa el resultado eventual de una tarea asincrónica.

1 Estados

Resolved (exitoso) o Rejected (erróneo).

2 Métodos

Utiliza `.then()`, `.catch()` y `.finally()` para manejar resultados.

Ejemplo de Promesa

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

```
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);
    })
}
```

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.

Async/Await



Async

Palabra clave para definir una función asíncrona.

Await

Espera la resolución de una promesa antes de continuar.

Try/Catch

Bloque recomendado para manejar errores en funciones async.

Fetch: consumiendo datos externos

Fetch: Consumiendo Datos Externos

API nativa para hacer solicitudes HTTP a servidores web

Ventajas

Reemplaza XMLHttpRequest,
utiliza promesas para manejar
respuestas.

Uso

Función global fetch()
devuelve una promesa con la
respuesta HTTP.



Fetch vs Axios

Fetch

Nativo, requiere configuración manual para algunas funcionalidades.

Axios

Biblioteca externa con características integradas y manejo automático de errores.

Conclusión y Próximos Pasos

Servidores Web

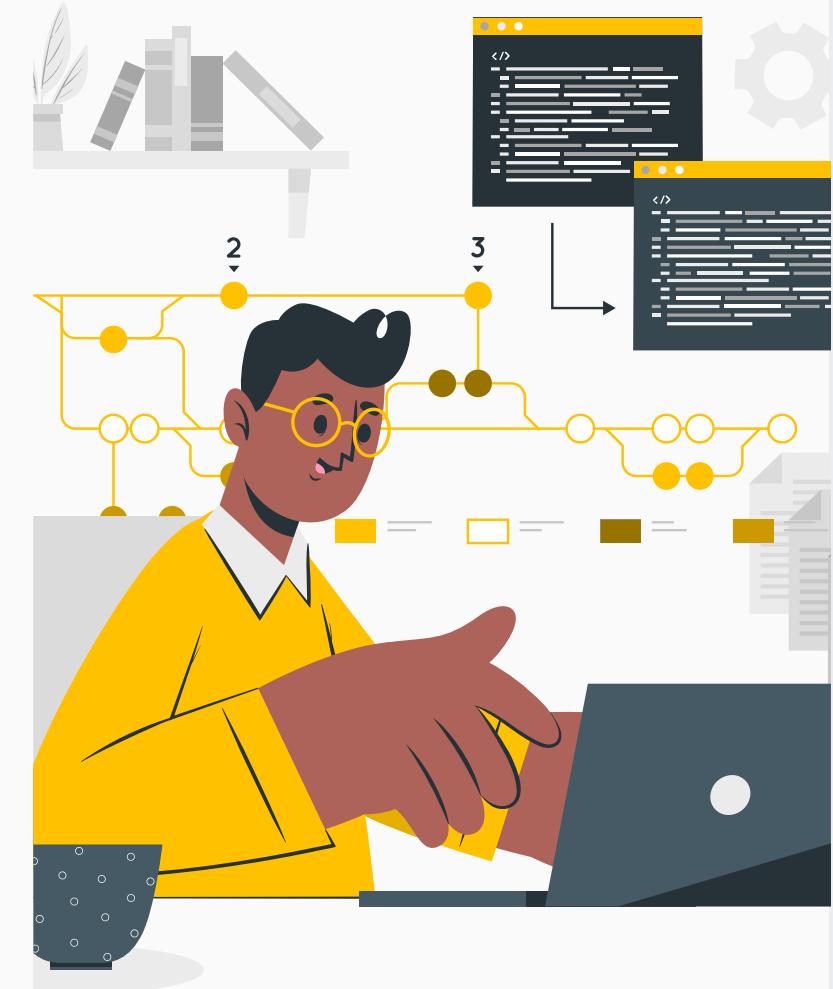
Comunicación web y funcionamiento de servidores.

Patrones de Arquitectura

Cimientos de los proyectos de programación.

Creando un Servidor Web

Primeros pasos en la creación de servidores con Node JS.



Materiales y recursos adicionales

-
- 1** Event Loop en Node JS - [Documentación Oficial](#)
 - 2** Aprende más sobre la API Fetch y sus capacidades: [MDN Web Docs – Fetch API](#)
 - 3** Una alternativa a Fetch para solicitudes HTTP avanzadas: [Axios GitHub Repository](#)





Ejercicio Práctico

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 manejas 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".



Ejercicio Práctico

Obligatorio

Misión 1:

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.



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”.



Ejercicio Práctico

Obligatorio

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".



¡Nuevo cuestionario en Campus!

-  No olvides que los cuestionarios son de carácter obligatorio para poder avanzar con la cursada.

