

Prompt 1

Estoy aprendiendo Javascript pero me cuesta trabajo comprender las funciones asíncronas. Puedes explicarlo de manera simple y con algún ejemplo que pueda implementarse fácilmente?

Respuesta 1

¡Claro! Vamos a ver **qué son las funciones asíncronas en JavaScript**, por qué existen y cómo usarlas con un ejemplo súper simple que puedes probar hoy mismo.

¿Qué es la asincronía en JavaScript?

JavaScript es un lenguaje **de un solo hilo**. Eso quiere decir que solo puede hacer *una cosa a la vez*.

Pero... ¿qué pasa cuando haces algo que tarda, como:

- Leer un archivo
- Consultar una API
- Esperar la respuesta de una base de datos
- Poner un temporizador

Si JavaScript esperara “bloqueado” hasta que eso termine, **tu programa se congelaría**.

Por eso existen las funciones **asíncronas**:

👉 Permiten que JavaScript “ponga a trabajar algo en segundo plano” mientras sigue ejecutando otras instrucciones.

Lo básico: **async** y **await**

- **async** convierte una función en asíncrona.

- `await` le dice a JavaScript: *espera a que esta operación termine...* pero sin bloquear todo lo demás.
-

Ejemplo muy simple: esperar datos “fingidos”

Imagina que quieres obtener datos de un servidor, pero tardan 2 segundos.

```
function obtenerDatos() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve("Datos del servidor");
    }, 2000);
  });
}

async function main() {
  console.log("Solicitando datos...");

  const resultado = await obtenerDatos(); // <-- Aquí esperamos la
  promesa

  console.log("Resultado recibido:", resultado);
}

main();
```

✓ ¿Qué pasa aquí?

1. `obtenerDatos()` regresa una **promesa** que se resuelve en 2 segundos.
2. `main()` es una función **async**, así que dentro se puede usar `await`.
3. Mientras se esperan los datos, JavaScript **no se bloquea**.
4. Cuando la promesa se resuelve, el valor se guarda en `resultado`.



Ejemplo aún más real: usar fetch

Si copias esto en tu navegador o en Node, funciona:

```
async function obtenerUsuario() {  
  const respuesta = await  
  fetch("https://jsonplaceholder.typicode.com/users/1");  
  const data = await respuesta.json();  
  console.log(data);  
}  
  
obtenerUsuario();
```

¿Qué hace?

1. Llama a una API falsa que siempre funciona.
2. `fetch` regresa una promesa.
3. `await` espera la respuesta.
4. Imprime el usuario obtenido.



Explicación en una frase

La asincronía permite que JavaScript inicie una tarea que tarda, siga trabajando en otras cosas, y luego retome esa tarea cuando termine.



Mini ejercicio para ti

Cambia el código para que:

- primero pida *usuario 1*

- luego *usuario 2*
- pero sin que se mezclen los prints
- usa dos `await`

Si quieres, te lo reviso.

Si quieres, también puedo explicarlo con una metáfora, diagramas o ayudarte a practicar más con ejercicios. ¿Quieres?

Prompt 2

Necesito llegar al mismo resultado del segundo ejemplo, pero sin usar `await`, `fetch` y `async`

Respuesta 2

Perfecto, podemos llegar **al mismo resultado** (obtener datos de una URL y mostrarlos) **sin usar `async`, `await` ni `fetch`**.

La forma clásica de hacerlo en JavaScript es con **`XMLHttpRequest`** o con **promesas manuales + callbacks**.

Aquí te dejo la versión más simple usando **`XMLHttpRequest`**, que existía antes de `fetch`.

Ejemplo equivalente usando `XMLHttpRequest` (sin `async`, `await` ni `fetch`)

```
function obtenerUsuario(callback) {  
  const xhr = new XMLHttpRequest();  
  
  xhr.open("GET", "https://jsonplaceholder.typicode.com/users/1");
```

```
xhr.onload = function() {
  if (xhr.status === 200) {
    const data = JSON.parse(xhr.responseText);
    callback(null, data); // éxito
  } else {
    callback("Error en la petición");
  }
};

xhr.onerror = function() {
  callback("Error de red");
};

xhr.send();
}

obtenerUsuario(function(error, usuario) {
  if (error) {
    console.log(error);
  } else {
    console.log("Usuario obtenido:", usuario);
  }
});
```

¿Qué está pasando aquí?

✓ **XMLHttpRequest** hace la petición HTTP

Es la manera antigua de pedir datos a un servidor.

✓ **No usamos `async` ni `await`**

Porque el código funciona mediante **callbacks**:

- La función `obtenerUsuario` NO regresa el resultado de inmediato.
- En su lugar, llama a `callback` cuando la petición termina.

✓ **`callback(error, data)`**

- Si hubo error → lo pasamos como primer parámetro.
 - Si todo salió bien → `error = null` y enviamos los datos.
-



Resultado

Obtendrás exactamente lo mismo que con `fetch`:

```
Usuario obtenido: { id: 1, name: "...", email: "...", ... }
```



¿Quieres otra versión?

Si quieres, puedo darte:

- Una versión usando **promesas sin async/await**
- Una versión usando **jQuery AJAX**
- Una versión usando **Node.js puro (http module)**
- Una explicación de cómo funciona **el event loop** cuando usas callbacks

Solo dime cuál te interesa.

Prompt 3

```
$ node snippet.js
C:\Users\HP\Desktop\Bootcamp Java\Javascript\JsIA\snippet.js:3
  const xhr = new XMLHttpRequest();
                ^
ReferenceError: XMLHttpRequest is not defined
```

Respuesta 3

Perfecto, esto tiene sentido!

El error ocurre porque **XMLHttpRequest** no existe en **Node.js**, solo existe en el navegador.

En Node.js, si no quieres usar `fetch`, `async` ni `await`, la alternativa clásica es usar el **módulo nativo `http`** (o `https`).

Aquí te dejo el equivalente usando **solo Node.js puro**, sin `async/await/fetch`:

✓ Ejemplo en Node.js usando `https` (sin `async`, `await` ni `fetch`)

```
const https = require("https");

function obtenerUsuario(callback) {
  https.get("https://jsonplaceholder.typicode.com/users/1", (res) => {
    {
      let data = "";

      // Se acumulan los chunks de datos
      res.on("data", (chunk) => {
        data += chunk;
      });

      // Cuando la respuesta termina:
      res.on("end", () => {
        try {
          const json = JSON.parse(data);
          callback(null, json);
        } catch (error) {
          callback("Error al parsear JSON");
        }
      });
    }
  }).on("error", (err) => {
    callback("Error de red: " + err.message);
  });
}

// Uso:
obtenerUsuario((error, usuario) => {
  if (error) {
    console.log("Error:", error);
  } else {

```

```
    console.log("Usuario obtenido:", usuario);  
  }  
});
```

¿Qué está pasando aquí?

✓ `https.get(...)`

Hace una petición GET sin bloqueos.

✓ Trabajamos con callbacks

No usamos `async/await` porque el callback recibe la respuesta cuando termina.

✓ Armado manual de la respuesta

En Node los datos llegan en fragmentos ("chunks"), por eso se concatenan.

✓ Al final:

Se parsea el JSON y se entrega vía callback.

Prompt 4

Podrías añadir comentarios para explicar lo que ocurre en cada línea del código

Respuesta 4

¡Claro! Aquí tienes el mismo ejemplo de Node.js usando `https`, pero **comentado línea por línea** para que entiendas exactamente qué está pasando.

Código explicado paso a paso (sin `async/await/fetch`)

```
// Importamos el módulo nativo de Node.js para hacer peticiones  
HTTPS
```



```
const https = require("https");

// Definimos una función que recibirá un callback cuando los datos
estén listos
function obtenerUsuario(callback) {

    // Realizamos una petición GET a la URL deseada
    https.get("https://jsonplaceholder.typicode.com/users/1", (res) =>
    {

        // Aquí guardaremos los fragmentos ("chunks") de la respuesta
        let data = "";

        // Cada vez que llega un fragmento de datos, se añade a la
variable "data"
        res.on("data", (chunk) => {
            data += chunk;
        });

        // Cuando la respuesta ha terminado de llegar
        res.on("end", () => {
            try {
                // Convertimos el texto recibido en un objeto JSON
                const json = JSON.parse(data);

                // Llamamos al callback indicando que NO hay error (primer
parámetro null)
                // y enviamos los datos parseados
                callback(null, json);
            } catch (error) {
                // Si falla el parseo, reportamos un error al callback
                callback("Error al parsear JSON");
            }
        });

        // Si la petición falla (problema de red, DNS, etc.)
    }).on("error", (err) => {
        callback("Error de red: " + err.message);
    });
}
```

```
// Llamamos a nuestra función y proporcionamos un callback
obtenerUsuario((error, usuario) => {

  // Si hubo error, lo mostramos
  if (error) {
    console.log("Error:", error);

  } else {
    // Si no hubo error, mostramos el usuario obtenido
    console.log("Usuario obtenido:", usuario);
  }
});
```



Explicación resumida

- Node.js no tiene **XMLHttpRequest**, así que usamos **https** para hacer peticiones.
 - La función **https.get** es **asíncrona**, por lo que no regresa los datos de inmediato.
 - La respuesta del servidor llega poco a poco → por eso usamos **res.on("data")**.
 - Cuando termina, convertimos el texto a JSON.
 - El resultado se entrega mediante un **callback**, que se ejecuta cuando la operación ya terminó.
-

Si quieres, puedo hacer una versión equivalente con:

- ✓ Promesas (sin `async/await`)
- ✓ Una simulación con **setTimeout** para comprender mejor la asincronía
- ✓ Un diagrama del flujo asíncrono
- ✓ Un ejemplo más sencillo

¿Quieres alguno?

Prompt 5

Puedes generar una infografía donde se explique lo que se hizo de manera sencilla y fácil de entender para principiantes en programación.



¿CÓMO FUNCIONA LA ASINCRONÍA EN JAVASCRIPT'?

Javascript es de un solo hilo

Eso quiere decir que solo puede hacer una cosa a la vez.



Entonces, ¿qué pasa con las tareas que tardan?

- Leer un archivo
- Consultar una API
- Esperar una respuesta de una base de datos
- Poner un temporizador



Javascript no se bloquea

Porque permite que la tarea trabaje en segundo plano y sigue con otras instrucciones.



EJEMPLO

```
function obtenerDatos() {  
  return new Promise(){  
    setTimeout(() => {  
      resolve(2900,  
        'datos del servidor'  
      )  
    }, 2000)  
  }  
}  
async function main()  
  console.log('Solicitando datos...')  
  const resultado = await obtenerDatos()  
  console.log('Resultado recibido: ' + resultado)  
}
```



Código Final

```
// Importamos el módulo nativo de Node.js para hacer peticiones HTTPS
const https = require("https");

// Definimos una función que recibirá un callback cuando los datos
estén listos
function obtenerUsuario(callback) {

    // Realizamos una petición GET a la URL deseada
    https.get("https://jsonplaceholder.typicode.com/users/1", (res) => {

        // Aquí guardaremos los fragmentos ("chunks") de la respuesta
        let data = "";

        // Cada vez que llega un fragmento de datos, se añade a la variable
"data"
        res.on("data", (chunk) => {
            data += chunk;
        });

        // Cuando la respuesta ha terminado de llegar
        res.on("end", () => {
            try {
                // Convertimos el texto recibido en un objeto JSON
                const json = JSON.parse(data);

                // Llamamos al callback indicando que NO hay error (primer
parámetro null)
                // y enviamos los datos parseados
                callback(null, json);
            } catch (error) {
                // Si falla el parseo, reportamos un error al callback
                callback("Error al parsear JSON");
            }
        });
    });

    // Si la petición falla (problema de red, DNS, etc.)
}).on("error", (err) => {
    callback("Error de red: " + err.message);
});
}
```

```
// Llamamos a nuestra función y proporcionamos un callback
obtenerUsuario((error, usuario) => {

    // Si hubo error, lo mostramos
    if (error) {
        console.log("Error:", error);
    } else {
        // Si no hubo error, mostramos el usuario obtenido
        console.log("Usuario obtenido:", usuario);
    }
});
```