ASINCRONISMO EN JAVASCRIPT

TEMA 02 – BLOQUEO VS NO BLOQUEO

2.1. Introducción al concepto de bloqueo

En el entorno de JavaScript, tanto en el lado del cliente (navegadores web) como en el lado del servidor (Node.js), es fundamental comprender las diferencias entre las operaciones que bloquean la ejecución y aquellas que no lo hacen. Este conocimiento resulta clave para desarrollar aplicaciones eficientes, escalables y con un rendimiento adecuado. A continuación, profundizaremos en los conceptos de sincronía/asíncronía y presentaremos ejemplos para ilustrar el impacto del bloqueo y la importancia de manejar correctamente las operaciones que pueden afectar al rendimiento de nuestras aplicaciones.

2.2. Diferencia entre operaciones síncronas y asíncronas

¿Qué se entiende por operaciones síncronas?

Las operaciones **síncronas** se llevan a cabo **secuencialmente**. Cuando el programa se encuentra con una operación síncrona, detiene su ejecución hasta que dicha operación ha concluido. Solo entonces pasa a la instrucción siguiente. En términos sencillos, el flujo del programa espera la finalización de una tarea antes de continuar.

Ventajas de las operaciones síncronas

• **Simplicidad**: El flujo de trabajo es más fácil de razonar, ya que cada instrucción se ejecuta en orden estricto y no hay “saltos” temporales.

• **Código lineal**: Debido a que no se delega nada a rutinas asíncronas, el código suele escribirse de manera más directa (un paso detrás de otro), facilitando la lectura en escenarios muy simples.

Desventajas de las operaciones síncronas

• **Bloqueo de la ejecución**: Si la operación lleva demasiado tiempo (por ejemplo, cálculos intensivos o acceso a recursos externos con

latencia, como consultas a bases de datos o peticiones de red), el hilo de ejecución principal permanecerá bloqueado. Esto implica que la aplicación dejará de responder mientras la operación no haya terminado.

• **Rendimiento limitado**: En entornos de alta concurrencia (varios usuarios, múltiples peticiones de datos o eventos), el bloqueo puede provocar tiempos de respuesta muy elevados, afectando negativamente la experiencia del usuario y el rendimiento global de la aplicación.

¿Qué se entiende por operaciones asíncronas?

Las operaciones **asíncronas** permiten que el flujo de **ejecución no se bloquee** a la espera de que una tarea termine. En JavaScript, cuando nos encontramos con una operación asíncrona, la enviamos a un segundo plano (por ejemplo, el sistema operativo o una API), y seguimos ejecutando el resto del código sin esperar el resultado inmediato.

Ventajas de las operaciones asíncronas

• **No bloqueo**: El flujo de la aplicación no queda paralizado. JavaScript continúa ejecutando otras partes del código y, cuando la operación asíncrona concluye, el resultado se gestiona mediante funciones de devolución de llamada (callbacks), promesas o async/await.

• **Escalabilidad**: Permite manejar más peticiones y más usuarios al mismo tiempo, ya que las tareas no se “estancan” esperando la finalización de un proceso costoso.

Desventajas de las operaciones asíncronas

• **Complejidad de gestión**: Requiere una buena gestión de callbacks, promesas o uso de async/await para evitar anidar demasiadas operaciones y complicar la lectura (lo que en la jerga a veces se llama *callback hell*).

• **Dificultad de depuración**: Cuando ocurren errores en hilos asíncronos o en promesas, rastrear el punto de fallo exacto puede ser menos intuitivo que en un código puramente secuencial.

2.3. Ejemplos

Ejemplo 1: Operación síncrona de bloqueo

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

// Simulamos una operación costosa con un bucle que tarde unos segundos
function operacionSincronaCostosa() {
  const start = Date.now();
  // Bucle de 2 segundos aproximadamente
  while (Date.now() - start < 2000) {
    // Simula una tarea muy pesada (cálculos intensivos)
  }
  console.log("Operación síncrona completada");
}
operacionSincronaCostosa();
console.log("Fin del programa");
```

1. El código escribe “Inicio del programa”.
2. Ejecuta la función *`operacionSincronaCostosa()`*, que **bloquea** el **hilo principal**, ya que el while impide que se ejecute cualquier otra instrucción hasta que se cumplan los 2 segundos de espera.
3. Tras finalizar la operación, se imprime *`“Operación síncrona completada”.`*
4. Finalmente, se escribe *`“Fin del programa”`*.

En este escenario, observamos claramente que el programa deja de responder durante esos 2 segundos, porque el **bucle** es **bloqueante** (hasta que no concluye, no se continúa con el flujo). Cualquier otra acción que intente realizar el usuario o el sistema tendrá que esperar.

Ejemplo 2: Operación asíncrona no bloqueante

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


function operacionAsincronaCostosa(callback) {
   // Usamos setTimeout para simular una operación costosa que complete en 2s
  setTimeout(() => {
    console.log("Operación asíncrona completada");
    callback();
  }, 2000);
}
operacionAsincronaCostosa(() => {
  console.log("Callback ejecutado tras la operación asíncrona");
});
console.log("Fin del programa");
```

1. El código escribe *“Inicio del programa”*.
2. Llama a *operacionAsincronaCostosa()*, que internamente utiliza *setTimeout* para esperar 2 segundos. Sin embargo, el hilo principal **no se bloquea**.
3. Inmediatamente se imprime *“Fin del programa”*, reflejando que la ejecución continúa sin esperar los 2 segundos.
4. Pasados los 2 segundos, se ejecuta el código dentro de la función pasada como callback: primero se imprime *“Operación asíncrona completada”* y, a continuación, *“Callback ejecutado tras la operación asíncrona”*.

Con este comportamiento, cualquier otra parte de nuestro programa permanece libre para seguir ejecutándose. No hay un bloqueo del hilo principal, de modo que si surge alguna otra tarea o acción de usuario, el programa podría gestionarla mientras espera la finalización de la operación costosa.

2.4. Conclusiones

La distinción entre bloqueo y no bloqueo es esencial en **JavaScript**, sobre todo porque el lenguaje **se ejecuta en un solo hilo**, **denominado *event loop***. Cualquier operación que bloquee este único hilo de ejecución retrasará el resto de la aplicación, perjudicando su rendimiento y la experiencia del usuario. Por el contrario, las operaciones asíncronas aprovechan la naturaleza del *event loop* para delegar tareas costosas y permitir que el código principal fluya sin interrupciones.

En el desarrollo de aplicaciones web, especialmente en entornos donde se gestionan numerosas peticiones y eventos, ya sea en el lado cliente o en el lado servidor, **es muy importante identificar y tratar debidamente todas las operaciones susceptibles de bloqueo**. Adoptar un enfoque asíncrono garantiza escalabilidad, mayor capacidad de respuesta y una experiencia más fluida para el usuario final.

Para finalizar, debe quedar claro que JavaScript se inclina hacia el paradigma de la asíncronía y cuáles son las razones técnicas para optar por operaciones que no bloqueen la ejecución.

A lo largo de los próximos documentos, iremos profundizando en las diferentes técnicas y patrones de programación asíncrona (callbacks, promesas, async/await) y en cómo utilizarlos de forma eficiente y limpia en nuestros proyectos.