ASINCRONISMO EN JAVASCRIPT

TEMA 03 – EVENT LOOP

3.1. Introducción al Event Loop

JavaScript es un lenguaje de programación orientado a eventos y basado en un modelo de concurrencia peculiar, definido por el llamado ***Event Loop***. Conocer en profundidad cómo funciona el Event Loop es fundamental para entender el comportamiento de las aplicaciones JavaScript, especialmente cuando ejecutan operaciones asíncronas, gestionan múltiples eventos o manipulan elementos en el navegador sin bloquear la interfaz de usuario.

En este tema, exploraremos la mecánica interna del ***Event Loop***, su relación con la ***Call Stack (***pila de llamadas) y la ***Task Queue*** (cola de tareas), así como el concepto de ***microtareas*** y ***macrotareas***, piezas clave para comprender cómo y cuándo se ejecutan las operaciones en JavaScript.

3.2. Event Loop: Call Stack y Task Queue

¿Qué es el Event Loop?

El *Event Loop* (bucle de eventos) es el **mecanismo interno que utiliza JavaScript para coordinar la ejecución del código**, **gestionar las operaciones asíncronas** y orquestar la r**espuesta a eventos** en un único hilo de ejecución.

A diferencia de lenguajes que permiten la ejecución paralela en varios hilos, JavaScript se ejecuta en un solo hilo, por tanto, cuando se presentan **tareas** que deben gestionarse de manera **asíncrona** (como peticiones a un servidor o la espera de un temporizador), el **Event Loop** se encarga de **programarlas**, **atenderlas** en el momento adecuado **y regresar** el control al flujo principal.

Para entender cómo funciona, tenemos que entender como funcionan en conjunto los siguientes elementos:

Call Stack (pila de llamadas):

Es el lugar donde se almacenan las funciones que se están ejecutando en un momento dado. Cuando se llama a una función, esta se introduce en la Call Stack (push); al terminar su ejecución, se elimina de la misma (pop). En JavaScript, solo hay un hilo de ejecución, por lo que solo puede procesarse una instrucción a la vez.

API Web / APIs del entorno:

Aunque no es parte directa del Event Loop, las APIs del navegador (o del lado del servidor) cumplen la función de gestionar ciertas operaciones externas al lenguaje. Por ejemplo, si pides un setTimeout, la API de temporizador del navegador es la que maneja la cuenta atrás y, al cumplirse el intervalo, envía la tarea a la cola correspondiente para que el Event Loop la recoja y la ejecute cuando sea apropiado.

Task Queue (cola de tareas):

También conocida como *Message Queue* o *Callback Queue*, aquí se almacenan las *tareas* (eventos o funciones de callback) que están listas para ejecutarse una vez que la Call Stack se haya vaciado o quede libre. Cuando el Event Loop detecta que la Call Stack está vacía, toma la primera tarea de la Task Queue y la introduce en la Call Stack para su ejecución.

El proceso se repite de manera **cíclica**: siempre que la Call Stack está libre, el Event Loop consulta la Task Queue, coge la siguiente tarea pendiente y la ejecuta. Por eso lo llamamos un bucle de eventos (event loop).

¿Cómo interactúan Call Stack y Task Queue?

Cuando se ejecuta código JavaScript de manera **sincróna** (por ejemplo, simples operaciones matemáticas, manipulaciones de variables, llamadas directas a funciones), todo transcurre en la ***Call Stack*** de forma **lineal**: las funciones entran y salen de la pila sin mayor complejidad.

Sin embargo, cuando se invoca una función **asíncrona** (por ejemplo, realizar una petición fetch para obtener datos del servidor), esa función registra la operación en la API externa (la del navegador o la del servidor). Después de iniciada la operación, la ejecución en la Call Stack continúa con las siguientes líneas de código, sin esperar de manera bloqueante. Una vez que la operación **asíncrona** haya **finalizado** (por ejemplo, el servidor devuelve la respuesta), la tarea resultante (el callback con los datos recibidos) se envía a la Task Queue. Cuando el Event Loop verifica que la Call Stack está libre, introduce el callback en la Call Stack, permitiendo la ejecución de la función con los datos solicitados.

Este ciclo de *poner tareas en cola* y *sacarlas para ejecutarlas* es el corazón del modelo de concurrencia de JavaScript.

Aunque el lenguaje esté restringido a un solo hilo de ejecución, esta organización basada en el Event Loop le permite reaccionar ante múltiples eventos y manejar asíncronamente multitud de operaciones, siempre y cuando la Call Stack no esté ocupada procesando otras tareas.

3.3. Microtareas y Macrotareas

El modelo de concurrencia de JavaScript se enriquece con la distinción entre **dos tipos de tareas que se colocan en diferentes colas**: las ***microtareas*** y las ***macrotareas***. Esta división influye directamente en el orden en el que se ejecutan las callbacks.

¿Qué son las macrotareas?

Las *macrotareas* (o “tareas normales”) incluyen la mayoría de las operaciones que generamos en JavaScript mediante:

- Eventos del DOM (por ejemplo, click, keydown, etc.).
- Operaciones asíncronas con temporizadores: setTimeout, setInterval.
- Tareas de I/O (Entrada/Salida), como peticiones AJAX o fetch.

• Llamadas a APIs externas del navegador (por ejemplo, geolocalización).

Cada vez que se completa una macrotarea y la Call Stack queda libre, el Event Loop revisa si hay microtareas pendientes antes de tomar la siguiente macrotarea. De esta manera, **las macrotareas se van atendiendo en orden, pero siempre dejando espacio para que, en cada iteración del bucle, las microtareas se procesen primero**.

¿Que son las microtareas?

Las *microtareas* tienen prioridad sobre las macrotareas. Se ejecutan antes de que el Event Loop tome la siguiente macrotarea de la cola de tareas. Entre las operaciones que generan microtareas, encontramos:

• **Promesas resueltas**: cuando se cumple una promesa, se planifica su callback (.then, .catch) en la cola de microtareas.

• **Mutaciones del DOM**, como los producidos por la API MutationObserver, que permite detectar cambios en la estructura de un documento, como la inserción o eliminación de nodos.

• **Funciones asíncronas con async/await** cuando se resuelve la promesa interna.

El flujo de ejecución es el siguiente:

1. Ejecutar la macrotarea actual (por ejemplo, el resultado de un setTimeout).

2. Finalizar la macrotarea y vaciar todas las microtareas pendientes de la cola de microtareas antes de continuar.

3. Cuando no queden microtareas pendientes, el Event Loop pasa a la siguiente macrotarea de la cola de tareas.

Ejemplo ilustrativo con microtareas y macrotareas

Supongamos que tenemos este código simplificado:

```
console.log("Inicio");
// Programamos una macrotarea
setTimeout(() => {
  console.log("Macrotarea: setTimeout");
}, 0);
// Generamos una promesa que se resuelve de inmediato
Promise.resolve()
  .then(() => {
    console.log("Microtarea: Promesa resuelta");
  });
console.log("Fin");
```

1. **Se ejecuta de forma sincrónica**:

• *`console.log("Inicio")`* → Imprime *“Inicio”*.

• *`setTimeout(...)`* se programa en la API del navegador y, en cuanto cumple con el tiempo (0 ms en este caso), se agenda en la cola de macrotareas.

• *`Promise.resolve()`* se resuelve inmediatamente y se agenda su *`callback (.then(...))`* en la cola de microtareas.

• *`console.log("Fin")`* → *Imprime “Fin”*.

2. **Fin de la tarea actual (el script)**:

• Ahora que el script principal ha terminado, la Call Stack está libre.

3. **Event Loop atiende la cola de microtareas**:

• Dado que hay una microtarea pendiente (la promesa resuelta), esta se ejecuta primero. Así, se imprime “Microtarea: Promesa resuelta”.

4. **Event Loop atiende la cola de macrotareas**:

• Tras vaciar la cola de microtareas, se toma la primera macrotarea pendiente (la del setTimeout). Imprime “Macrotarea: setTimeout”.

La secuencia real de salida en consola es:

1. Inicio

2. Fin

3. Microtarea: Promesa resuelta

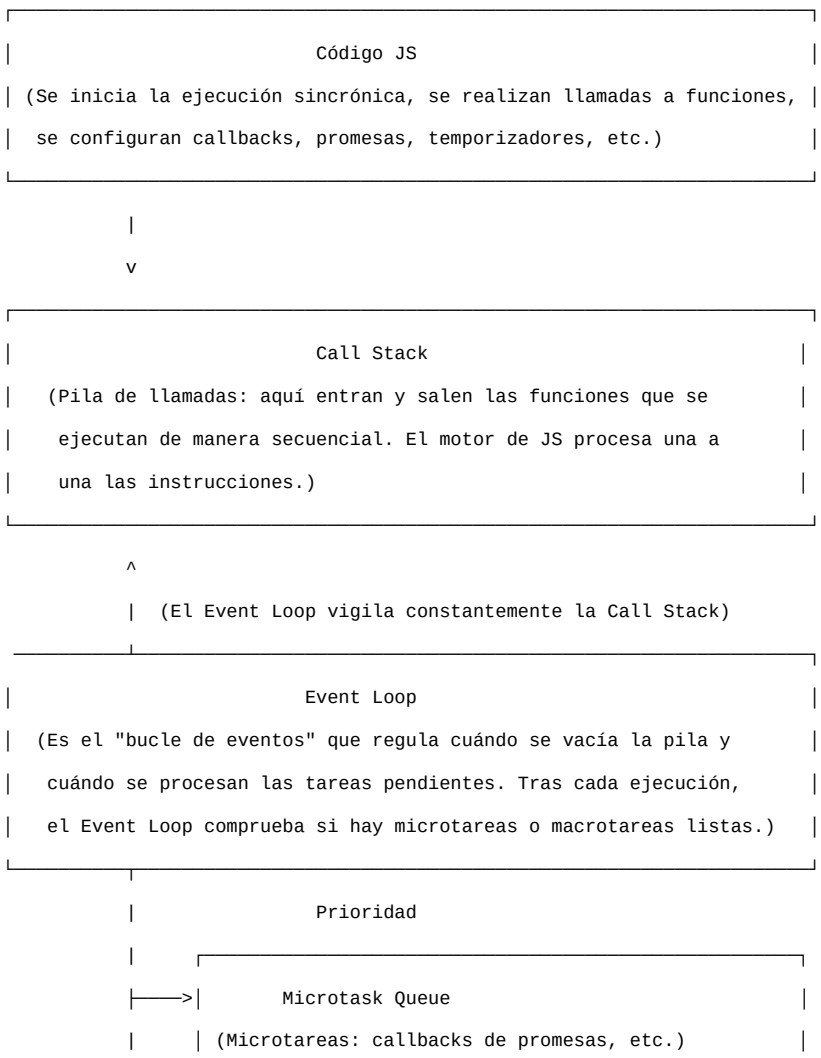
4. Macrotarea: setTimeout

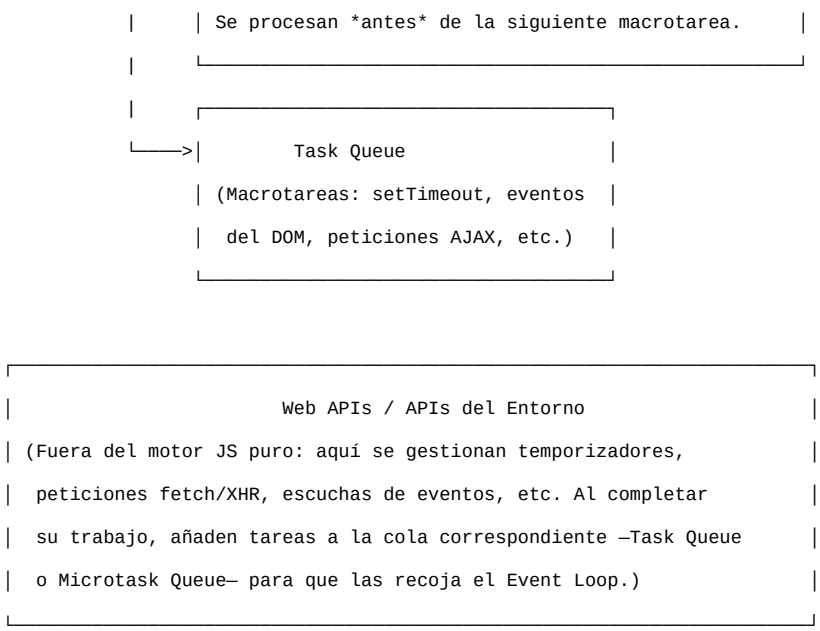
3.4. Representaciones gráficas del proceso

Esquema 1

A continuación, se presenta una representación gráfica que muestra de forma simplificada la relación entre la **Call Stack**, el **Event Loop**, la **Task Queue** y los dos tipos de tareas (**macrotareas** y **microtareas**). Asimismo, se ilustra el papel de las **Web APIs** (o APIs del entorno), que actúan fuera del motor de JavaScript pero cooperan para manejar las operaciones asíncronas.

Cada bloque representa un “espacio” donde ocurren procesos concretos. Las flechas indican el flujo de tareas:





Código JS:

Aquí se lee y ejecuta el script principal de forma síncrona. Si hay llamadas a funciones, promesas, setTimeout, etc., se registran en las **Web APIs** o se añaden posteriormente a las colas correspondientes.

Call Stack (Pila de Llamadas):

Todas las funciones que se están ejecutando en un momento dado van pasando por la pila. Cuando invocas una función, esta se apila y, al terminar, se desapila. JavaScript es monohilo , por lo que únicamente puede procesar una cosa a la vez en esta pila.

Event Loop (Bucle de Eventos):

Es un proceso que constantemente revisa si la **Call Stack** está vacía. Si lo está, el Event Loop comprueba primero si existen **microtareas** pendientes en la **Microtask Queue**. De haberlas, las ejecuta.

Después de procesar todas las microtareas, toma la siguiente **macrotarea** de la **Task Queue** y la coloca en la **Call Stack**.

Microtask Queue:

Aquí se encolan las microtareas, que tienen prioridad sobre las macrotareas. Principalmente se generan al resolverse promesas (Promise.resolve(), métodos .then(), async/await) y ciertos cambios internos del DOM.

Antes de procesar la siguiente macrotarea, JavaScript vacía por completo la cola de microtareas.

Task Queue (cola de Macrotareas):

Incluye las tareas relacionadas con setTimeout, setInterval, peticiones AJAX o fetch, e incluso eventos del DOM (click, submit, etc.). Tras cada ciclo del Event Loop, una vez procesadas las microtareas, se atiende la siguiente macrotarea.

Web APIs / **APIs del entorno**:

No forman parte directamente del motor JavaScript, sino que son utilidades ofrecidas por el entorno (un navegador o Node.js). Por ejemplo, cuando llamas a setTimeout, la cuenta atrás la gestiona el navegador. Una vez cumplido el tiempo, la API añade la callback a la **Task Queue** (cola de macrotareas) para que el Event Loop la ejecute cuando la Call Stack esté libre.

Resumen de ideas clave

• La *Call Stack* ejecuta las funciones una detrás de otra.

• Las *Web APIs* (en un navegador) y las APIs del entorno (en Node.js) gestionan las operaciones asíncronas.

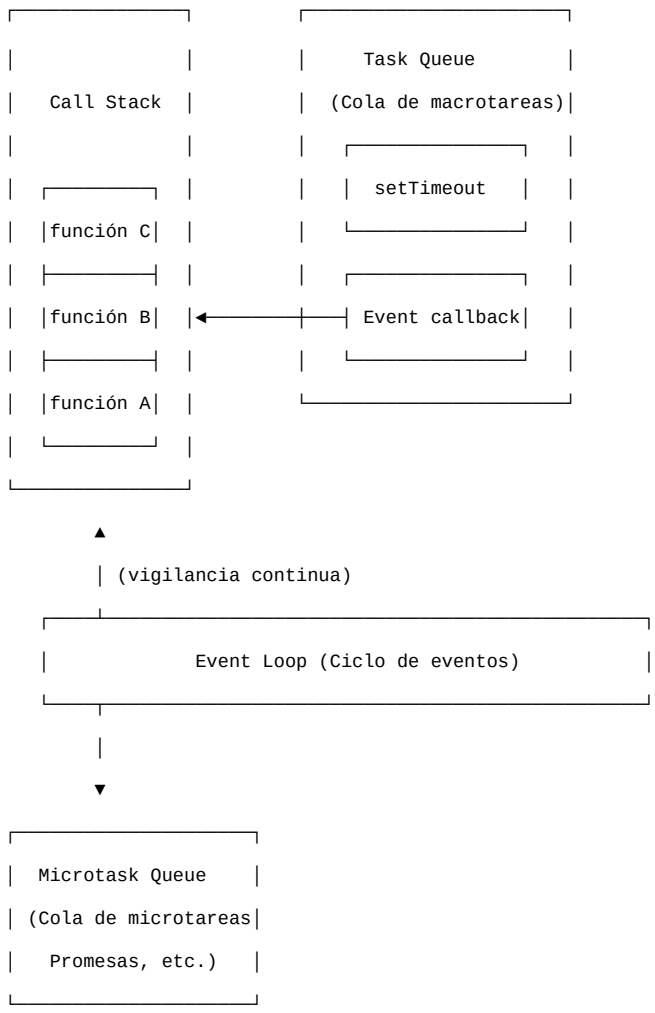
• Cuando acaban, dichas operaciones colocan sus callbacks en la Task Queue o la Microtask Queue.

• El *Event Loop* controla el orden: primero ejecuta todo lo que haya en la Microtask Queue (si hay microtareas pendientes), y luego pasa a tomar la siguiente macrotarea en la Task Queue.

• Este mecanismo permite que JavaScript maneje de forma asíncrona múltiples eventos y tareas sin bloquear el hilo principal, pese a ser un lenguaje monohilo.

Esquema 2

Otra forma de ver estos procesos de interacción es la que se muestra a continuación



Call Stack (Pila de llamadas)

La **Call Stack** es el corazón de la ejecución secuencial en JavaScript. En ella se apilan y desapilan las funciones conforme el flujo de ejecución avanza.

En el diagrama, la Call Stack se representa como una columna en la que pueden encontrarse varias funciones (A, B, C, etc.) según vayan

siendo llamadas. Solo podemos procesar una tarea a la vez dentro de la Call Stack.

Task Queue (Cola de macrotareas)

La **Task Queue** (también conocida como *Message Queue*) es donde se encolan las *macrotareas*.

En el diagrama, dentro de la **Task Queue**, se observan ejemplos de tareas como setTimeout y Event callback. Cuando dichas tareas finalizan su gestión en las Web APIs (por ejemplo, el temporizador cuenta el tiempo y, al llegar a cero, empuja la callback a la cola), se colocan en la Task Queue.

Microtask Queue (Cola de microtareas)

La **Microtask Queue** es donde se almacenan las *microtareas*, que suelen tener prioridad sobre las macrotareas.

Entre las principales fuentes de microtareas destacan las promesas resueltas: cuando llamamos a `Promise.resolve()` o en cuanto se cumple la promesa en una llamada `fetch` con `.then()`, el callback se agenda en la cola de microtareas. También las operaciones internas de mutación del DOM, en ciertos contextos, se tratan como microtareas.

En el diagrama, este bloque se dibuja en la parte inferior, indicando que, cuando el Event Loop detecta microtareas pendientes, las ejecuta *antes* de pasar a la siguiente macrotarea.

Event Loop (Ciclo de eventos)

El **Event Loop** es un proceso que supervisa de manera continua el estado de la Call Stack y de las colas de tareas (principalmente, la Task Queue y la Microtask Queue).

Su dinámica de trabajo es la siguiente:

1. **Observa la Call Stack**. Si la Call Stack está vacía o acaba de terminar de procesar la función en curso, el Event Loop revisa qué tareas están pendientes.

2. **Revisa primero la Microtask Queue**: si hay microtareas, las ejecuta todas en orden hasta que la cola de microtareas quede vacía.

3. **Pasa a la Task Queue**: si ya no hay microtareas pendientes, toma la primera macrotarea de la Task Queue (si existe) y la introduce en la Call Stack para su ejecución.

4. **Se repite el ciclo**: una vez procesada la macrotarea, vuelve a revisar la Microtask Queue y repite el procedimiento.

Dicho de otra forma, **cada vez que la Call Stack se vacía**, se verifica primero si hay microtareas en la Microtask Queue (ya que tienen prioridad). Una vez gestionadas, se atiende la siguiente macrotarea en la Task Queue.

Interacción con las APIs del navegador (o APIs de Node.js)

Aunque no aparece explícito en el diagrama, es importante señalar que buena parte del trabajo asíncrono se inicia a través de las APIs del entorno (por ejemplo, en un navegador, la API de setTimeout, fetch, eventos del DOM, etc.).

Al completarse la operación asíncrona (por ejemplo, si se cumplen los milisegundos en setTimeout o llega la respuesta de un servidor con fetch), las APIs externas añaden el callback correspondiente a la cola apropiada (Task Queue o Microtask Queue).

3.5. Ejemplos

Ejemplo 1: Call Stack

La **Call Stack** registra en qué orden se invocan las funciones y cómo estas van entrando y saliendo de la pila. En un entorno monohilo como JavaScript, solo podemos ejecutar una función a la vez.

```
// Ejemplo 1: Call Stack básico
function funcionC() {
  console.log("funcionC: Ha sido llamada y está en la cima de la Call Stack");
}
function funcionB() {
  console.log("funcionB: Se llama a funcionC");
  funcionC();
  console.log("funcionB: Ha regresado de funcionC");
}
function funcionA() {
  console.log("funcionA: Se llama a funcionB");
  funcionB();
  console.log("funcionA: Ha regresado de funcionB");
}
console.log("Inicio de la ejecución principal ...");
funcionA();
console.log("Fin de la ejecución principal");
```

Explicación

1. El flujo comienza por la línea *`console.log("Inicio de la ejecución principal...")`*.

2. Llamamos a *`funcionA()`*, que se **apila** en la Call Stack.

3. Dentro de *`funcionA()`*, se llama a *`funcionB()`*, que entra en la Call Stack por encima de *`funcionA()`*.

4. Dentro de *`funcionB()`*, se llama a *`funcionC()`*, que se apila sobre todas las anteriores.

5. Cuando *`funcionC()`* termina, se desapila y regresa el control a *`funcionB()`*, que posteriormente se desapila y regresa el control a *`funcionA()`*.

6. Finalmente, *`funcionA()`* se desapila y volvemos al ámbito global, donde ejecutamos el último *`console.log`*.

Ejemplo 2: Macrotareas

Las macrotareas se generan cuando utilizamos, por ejemplo, *`setTimeout`*, *`setInterval`* o eventos del DOM. Una vez la operación asíncrona está lista, su callback se encola en la Task Queue, esperando a que el Event Loop la pase a la Call Stack cuando esta esté libre.

```
// Ejemplo 2: Macrotarea con setTimeout
console.log("1) Inicio del script");
setTimeout(() => {
      console.log("3) Esta es la macrotarea (setTimeout) ejecutándose");
}, 0);
console.log("2) Fin del script (código sincrónico)");
```

Explicación

1. Se ejecuta primero todo el código sincrónico: imprime “**1) Inicio del script”** y luego “**2) Fin del script”**.

2. setTimeout programa una macrotarea con retardo 0. Sin embargo, **esto no significa ejecución inmediata**, sino que se coloca en la cola de macrotareas.

3. Cuando la Call Stack está libre (tras el paso 2), el Event Loop recoge la macrotarea de setTimeout y ejecuta el callback, que imprime “**3) Esta es la macrotarea (setTimeout) ejecutándose”**.

El resultado en consola es:

```
1) Inicio del script
2) Fin del script
3) Esta es la macrotarea (setTimeout) ejecutándose
```

Ejemplo 3: Microtareas

Las microtareas suelen provenir de promesas resueltas. Tienen prioridad sobre las macrotareas: una vez se vacía la Call Stack, se procesan todas las microtareas pendientes antes de atender la siguiente macrotarea.

```
// Ejemplo 3: Microtareas con Promesas
console.log("A) Inicio sincrónico");
Promise.resolve()
  .then(() => {
    console.log("B) Promesa resuelta (microtarea 1)");
  })
  .then(() => {
        console.log("C) Encadenada tras la promesa anterior (microtarea 2)");
  });
console.log("D) Fin sincrónico");
```

Explicación

1. Se imprimen en primer lugar “**A) Inicio sincrónico”** y “**D) Fin sincrónico”**, porque las promesas se resuelven de manera asíncrona.

2. Una vez el script principal termina (la Call Stack se vacía), el Event Loop atiende la **cola de microtareas**. La promesa ya está resuelta, así que se ejecuta la primera *`callback (console.log("B) Promesa resuelta..."))`*.

3. Inmediatamente después, se encadena la siguiente microtarea (.then(...)), que imprime “**C) Encadenada tras la promesa anterior…”**.

El orden de salida en la consola es:

```
A) Inicio sincrónico
D) Fin sincrónico
B) Promesa resuelta (microtarea 1)
C) Encadenada tras la promesa anterior (microtarea 2)
```

3.6. Conclusión y relevancia en la práctica

Comprender esta arquitectura interna —monohilo con un *Event Loop* que coordina colas de macrotareas y microtareas— ayuda a explicar por qué JavaScript no se bloquea al esperar respuestas de un servidor o por un temporizador: esas operaciones se “desplazan” fuera del motor principal y se reinsertan como tareas cuando están listas.

Asimismo, es crucial para entender el orden de ejecución de callbacks, especialmente con promesas y async/await, y para evitar comportamientos inesperados (por ejemplo, por qué cierto console.log aparece antes que otro en la consola, a pesar de que su línea de código está aparentemente después).

Finalmente, los diagramas y ejemplos mostrados evidencian la relación entre la **Call Stack**, la **Task Queue**, la **Microtask Queue** y el **Event Loop**, pilares fundamentales de la programación asíncrona en JavaScript. Cuando se dominan estos conceptos, resulta más sencillo identificar y predecir el flujo de ejecución del código, optimizar la respuesta de una aplicación y solucionar problemas o cuellos de botella que puedan surgir en el desarrollo de proyectos web.