# ASINCRONISMO EN JAVASCRIPT

## Asincronismo

Capacidad de un programa para ejecutar ciertas tareas en segundo plano, permitiendo que **el flujo principal continúe con la ejecución de otras instrucciones sin tener que detenerse** hasta que se complete la tarea en cuestión.

*habitualmente Javascript ejecuta su código **de forma síncrona**, es decir en el orden natural al que estamos acostumbrados (de arriba a abajo, exceptuando en las llamadas a funciones). En presencia de operaciones de E/S (Entrada/Salida) o conexiones de red que requieren tiempo, **el programa queda bloqueado** hasta obtener los resultados.

## Event Loop (el director de la orquesta)

Mecanismo interno que permite coordinar las tareas asíncronas en JavaScript. Este bucle **recibe y gestiona los eventos y tareas pendientes de ser procesados**.

1. Se mantiene una **cola de tareas** en la que se registran todos los callbacks, promesas u operaciones asíncronas que están pendientes.

2. JavaScript ejecuta, **en primer lugar, el código principal de manera síncrona** hasta que se completa o encuentra una operación asíncrona.

3. Cuando se desencadena una **operación asíncrona** esta **se delegará al entorno externo** (como las APIs del navegador).

4. Mientras tanto, **el flujo principal no se bloquea**.

5. Una vez finalizada la operación asíncrona, se añade a la cola de tareas. El Event Loop comprueba si el hilo principal está libre y, cuando lo está, retoma la tarea pendiente, ejecutando el callback o resolviendo la Promesa.

## Call Stack (pila de llamadas)

Lugar en donde se almacenan las funciones que se están ejecutando en un momento dado. Cuando llaman a una función, esta se introduce (push) y al terminar, se elimina de la pila (pop). Como en JS solo hay 1 hilo de ejecución estas siempre se ejecutan de 1 en 1.

## API Web / API de entorno (los recaderos)

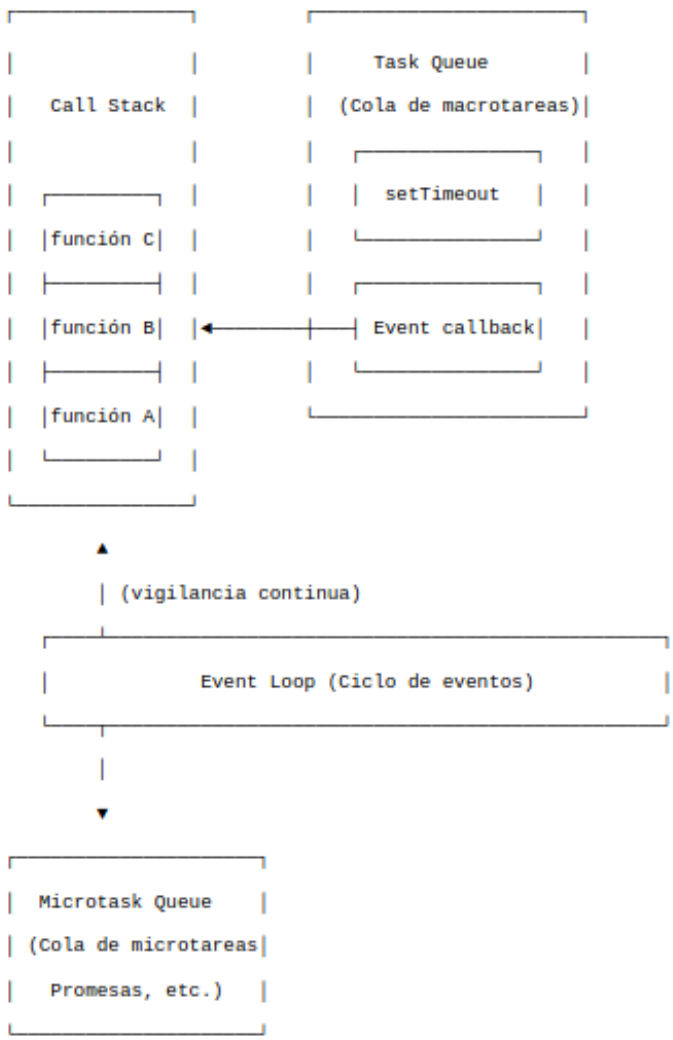
No forman parte del Event Loop. Cumplen ciertas operaciones externas al lenguaje como el temporizador de los setTimeout(), por ejemplo.

## Task Queue (cola de tareas)
Almacena las tareas, eventos o funciones de callback listas para ejecutarse cuando la Call Stack esté libre.

### Macrotareas (tareas normales)
Son la mayoría:
- Eventos del DOM.
- Operaciones asíncronas como setTimeout() y setInterval().
- Tareas I/O, como peticiones AJAX o fetch.
- Llamadas a APIs externas del naveagor (geolocalización).

### Microtareas
Tienen prioridad sobre las macrotareas, se ejecutan antes de que el Event Loop tome la siguiente macrotarea de la Task Queue. Ejemplos:
- El return de una Promise.
- Mutaciones en el DOM (adición o eliminación de nodos).
- Resoluciones de funciones asíncronas con async/await.

Un ejemplo de flujo de ejecución sería:
1. Macrotarea
2. Al finalizar esta, se vacían todas las microtareas pendientes.
3. El Event Loop pasa la siguiente macrotarea de la Task Queue (a la Call Stack).



## Promesas

Es un objeto que representa un valor que puede estar disponible ahora, en el futuro o nunca. Modela un flujo asíncrono para procesar el resultado de una operación de cierta latencia sin bloquear el resto de la ejecución.

```
const miPromesa = new Promise((resolve, reject) => {
     // Aquí va la operación asíncrona o un bloque de código
});
```

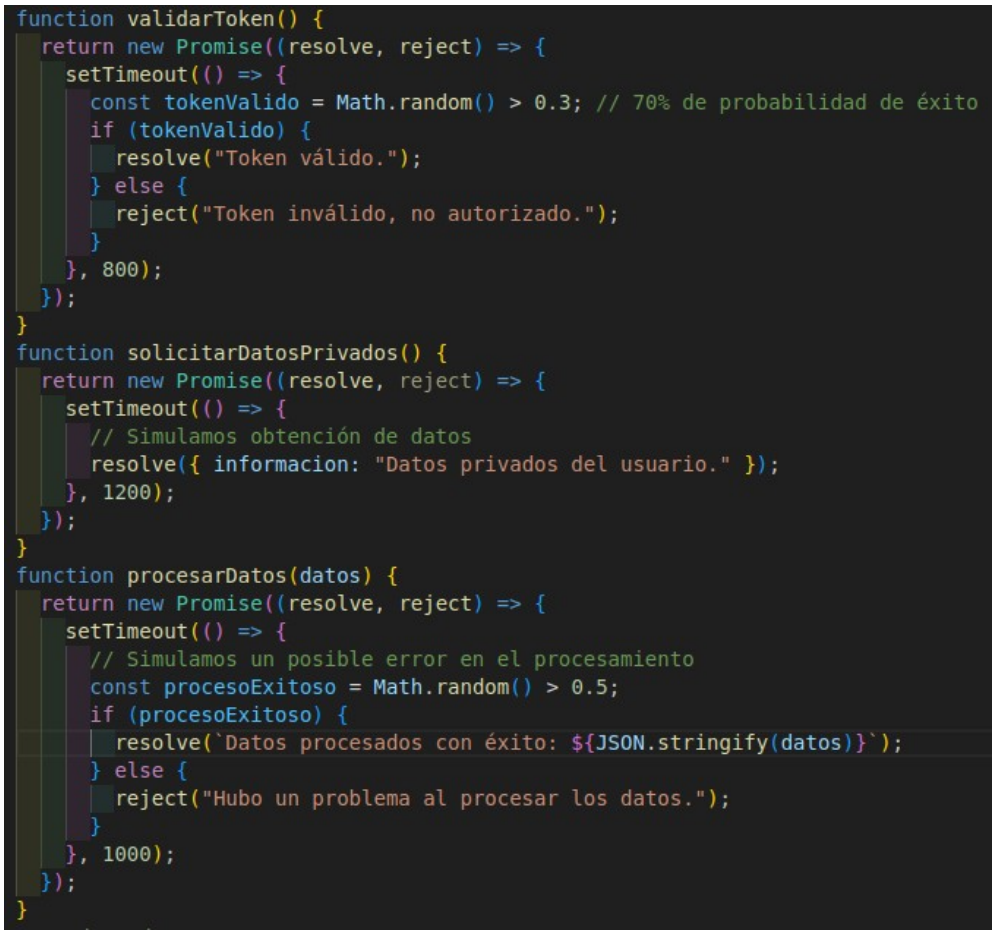
· *`resolve(valor)`* se invoca cuando la operación asíncrona ha tenido éxito o ha finalizado de forma satisfactoria, donde valor es el resultado que la promesa entrega.

· *`reject(error)`* se invoca cuando la operación ha fallado o se produce algún tipo de error, donde error describe la causa del fallo.

Estados básicos:
1. *Pending*: en proceso de ser resuelta.
2. *Fulfilled*: resultado exitoso.
3. *Rejected*: resultado fallido.

Las promesas se consumen encadenando .then(), .catch() y .finally() de manera que una vez se resuelve, el resultado deriva en true/false (*then* y *catch* respectivamente) y finalmente *finally.*

**Async/Await:** forma mas reciente e intuitiva de manejar operaciones asíncronas.



```
function validarToken() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const tokenValido = Math.random() > 0.3; // 70% de probabilidad de éxito
      if (tokenValido) {
        resolve("Token válido.");
      } else {
        reject("Token inválido, no autorizado.");
      }
    }, 800);
  });
}
function solicitarDatosPrivados() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // Simulamos obtención de datos
      resolve({ informacion: "Datos privados del usuario." });
    }, 1200);
  });
}
function procesarDatos(datos) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // Simulamos un posible error en el procesamiento
      const procesoExitoso = Math.random() > 0.5;
      if (procesoExitoso) {
        resolve(`Datos procesados con éxito: ${JSON.stringify(datos)}`);
      } else {
        reject("Hubo un problema al procesar los datos.");
      }
    }, 1000);
  });
}
```

```
// Cadena de Promesas
validarToken()
  .then((mensajeToken) => {
    console.log(mensajeToken);
    // Si la validación fue exitosa, solicitamos datos privados
    return solicitarDatosPrivados();
  })
  .then((datosPrivados) => {
    console.log("Datos privados obtenidos:", datosPrivados);
    // Encadenamos el procesamiento de datos
    return procesarDatos(datosPrivados);
  })
  .then((resultadoProcesado) => {
    console.log(resultadoProcesado);
  })
  .catch((error) => {
    // Cualquier error que ocurra en la cadena es capturado aquí
    console.error("Ha ocurrido un error en alguna parte del proceso:", error);
  })
  .finally(() => {
    console.log("Proceso de validación y procesamiento finalizado.");
  });
```