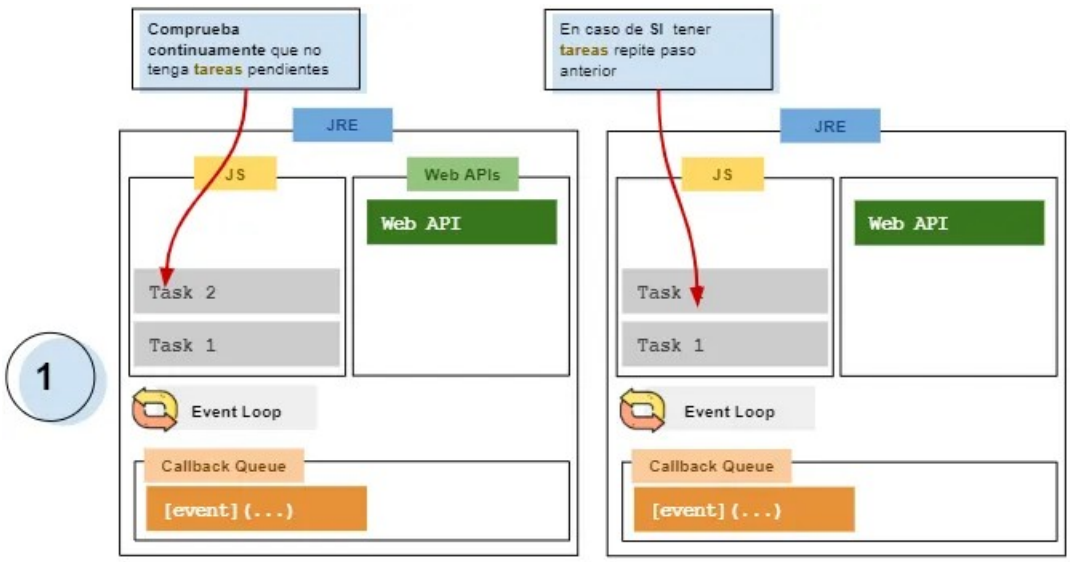
Curso de Asincronismo con JavaScript

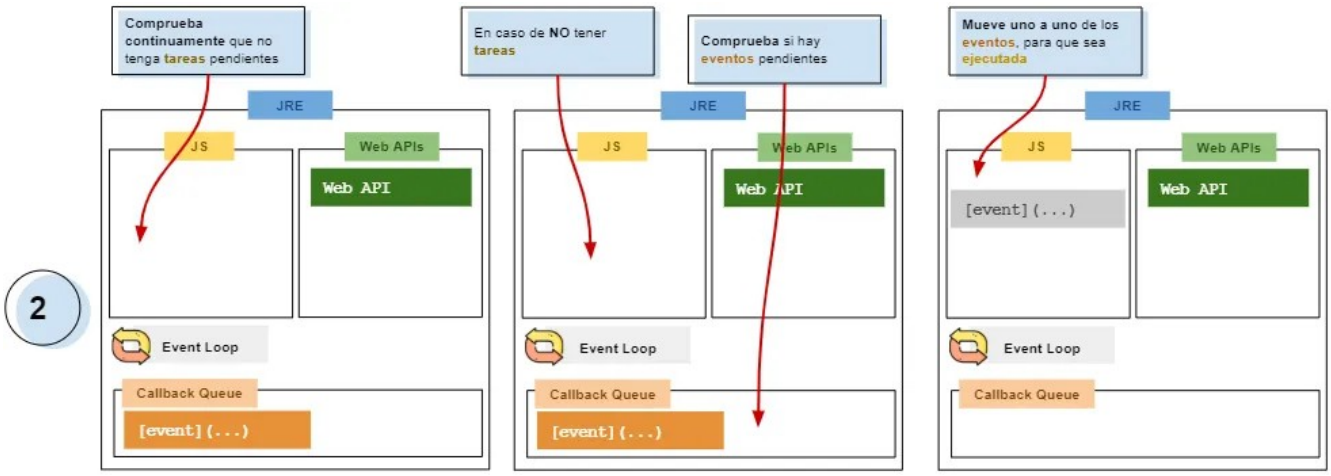
- JavaScript es single-threaded. Aún con múltiples procesadores, solo puede ejecutar tareas en un solo hilo, llamado el hilo principal.
- JavaScript es síncrono y no bloqueante, con un bucle de eventos (concurrencia), implementado con un único hilo para sus interfaces de I/O (input/output).
- **Bloqueante:** una tarea no devuelve el control de la implementación hasta que se ha completado.
- **No bloqueante:** Una tarea se devuelve inmediatamente con independencia del resultado. Si se completó, devuelve los datos. Si no, un error.
- **Síncrono:** Las tareas se ejecutan de forma secuencial, deben esperar a que se complete para continuar con la siguiente tarea.
- **Asíncrono:** Las tareas pueden ser realizadas más tarde, lo que hace posible que una respuesta sea procesada en diferido.
- **Concurrencia en JavaScript:** Utiliza un modelo de concurrencia basado en un "*loop de eventos*".
- **EventLoop:** El bucle de eventos es un patrón de diseño que espera y distribuye eventos o mensajes en un programa.
- **Callbacks:** Una función que es pasada como argumento de otra función y que será invocada.
- **Promesas (ES6):** Función no-bloqueante y asíncrona la cual puede retornar un valor ahora, en el futuro o nunca.
- **Async / Await (ES2017):** Permite estructurar una función asíncrona sin bloqueo de una manera similar a una función síncrona ordinaria.
- Después de cómo ha ido evolucionando podemos poner otra definición de **JavaScript**: Es asíncrono y no bloqueante, con un bucle de eventos (concurrencia) implementado con un único hilo para sus interfaces de I/O.

Event Loop: El componente Event loop tiene dos funciones principales:

1. Comprobar continuamente si el motor de JavaScript en el contexto de ejecución (Call stack) tiene tareas, en caso de SÍ tener tareas vuelve a comprobar.



2. Cuando el motor de JS NO tiene tareas, va a comprobar si hay eventos (cola de mensajes) en el Callback Queue, en caso de que tenga los manda al motor de JS a que los ejecute.



Importante: El paso de **callback queue** a el motor de JavaScript, lo hace uno a uno, esto quiere decir que lo hace de manera **síncrona**, por lo que debemos cuidar que el código ejecutado en el motor no sea bloqueante.

Las APIs cuando son requeridas (puede ser un evento como click, o una recuperación de respuesta como callback), se van apilando en el componente **callback queue**. Utiliza el modelo de **FIFO** (primera entrada, primera salida). Por lo que mantiene el orden en que fueron agregadas las tareas, y finalmente, es ejecutado en el componente del motor de JavaScript (Call stack).

En este ejemplo vamos a usar promesas para enfocarnos en cómo funciona el **JavaScript Runtime Environment.**

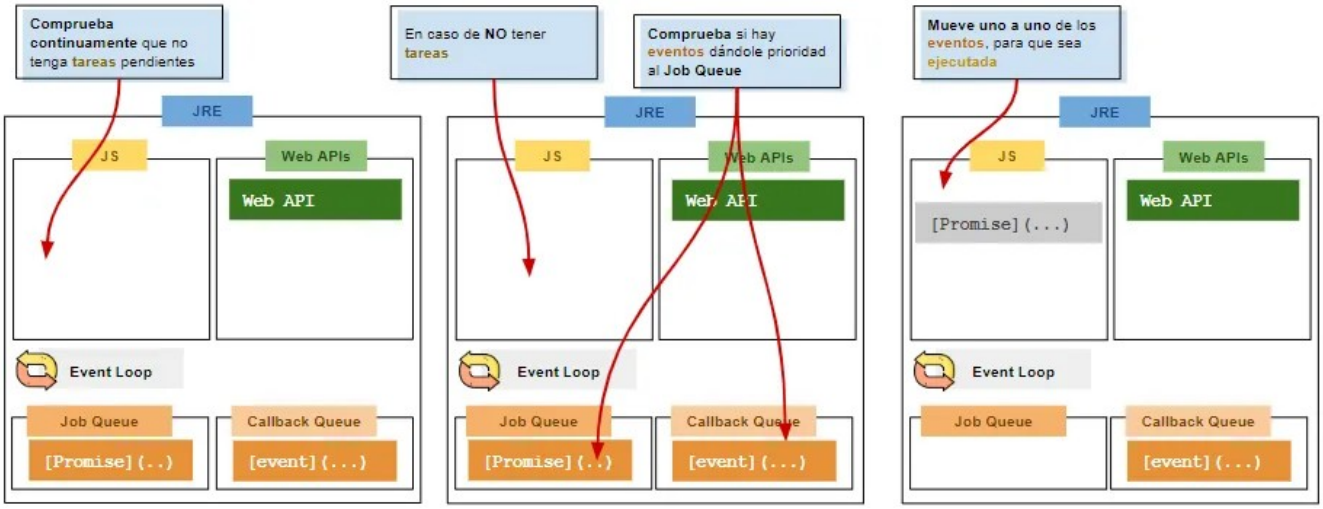
```
console.log('Hello!');

setTimeout(function timeout3000(){
    console.log('Hello from the setTimeOut => 3000ms');
}, 3000);

setTimeout(function timeout0(){
    console.log('Hello from the setTimeOut => 0ms');
}, 0);

for(let x=0; x <= 5 ; x++) {
    console.log('Hello for!!', x);
}

let promise = new Promise(function(resolve, reject) {
    resolve();
});

promise.then(function(resolve) {
    console.log('Hello from the Promise => 1st');
})
.then(function(resolve) {
    console.log('Hello from the Promise => 2nd');
});

console.log('Goodbye!!');
```

El resultado que sale en consola es el siguiente:

```
Hello!
Hello for!! 0
Hello for!! 1
Hello for!! 2
Hello for!! 3
Hello for!! 4
Hello for!! 5
Goodbye!!
Hello from the Promise => 1st
Hello from the Promise => 2nd
undefined
Hello from the setTimeOut => 0ms
Hello from the setTimeOut => 3000ms
```

Podemos observar que primero van las promesas, y al final los setTimeOut; el “culpable” de que salga en ese orden, es el componente **Job Queue**…

Cuando el motor de JavaScript no tiene tareas, el event loop le va a dar prioridad al componente de Job queue (al resultado de funciones asíncronas) y después al callback queue.



- **Memory Heap:** Los objetos son asignados a un montículo (espacio grande en memoria no organizado).
- **Call Stack:** Apila de forma organizada las instrucciones de nuestro programa. Aquí aplica LIFO (Last-in, First-out).
- **Task Queue:** Cola de tareas, se maneja la concurrencia, se agregan las tareas que ya están listas para pasar al stack (pila). El stack debe de estar vacío.
- **MicroTask Queue:** Las promesas tienen otra forma de ejecutarse y una prioridad superior.
- **Web APIs:** API stands for Application Programming Interface. A Web API is an application programming interface for the Web. APIs are constructs made available in programming languages to allow developers to create complex functionality more

easily. They abstract more complex code away from you, providing some easier syntax to use in its place.

- **Browser APIs:** constructs built into the browser that sits on top of the JavaScript language and allows you to implement functionality more easily.
- **Third-party APIs:** Constructs built into third-party platforms (e.g. Twitter, Facebook) that allow you to use some of those platform's functionality in your own web pages (for example, display your latest Tweets on your web page).

- **JavaScript del lado del cliente:**
  - setTimeout
  - XMLHttpRequest
  - File Reader
  - DOM
- **Node:**
  - fs
  - https

XMLHttpRequest: Es un objeto nativo del navegador que permite hacer solicitudes HTTP desde JavaScript. A pesar de tener la palabra “XML” en su nombre, se puede operar sobre cualquier dato, no solo en formato XML. Podemos cargar y descargar archivos, dar seguimiento y mucho más.
XMLHttpRequest tiene dos modos de operación: sincrónica y asíncrona.

fetch es el nombre de una nueva API para Javascript con la cuál podemos realizar peticiones HTTP asíncronas utilizando promesas y de forma que el código sea un poco más sencillo y menos verbose. La forma de realizar una petición es muy sencilla, básicamente se trata de llamar a fetch y pasarle por parámetro la URL de la petición a realizar.
El **fetch()** devolverá una **promesa** que será aceptada cuando reciba una respuesta y sólo será rechazada si hay un fallo de red o si por alguna razón no se pudo completar la petición.
El modo más habitual de manejar las promesas es utilizando .then().

Función async: La declaración de función async define una función asíncrona, la cual devuelve un objeto **AsyncFunction** (valor de retorno). Un objeto AsyncFunction, que representa una función asíncrona que ejecuta el código contenido dentro de la función.

Cuando se llama a una función async, esta devuelve un elemento Promise. Cuando la función async devuelve un valor, Promise se resolverá con el valor devuelto. Si la función async genera una excepción o algún valor, Promise se rechazará con el valor generado.

Una función async puede contener una expresión **await**, la cual pausa la ejecución de la función asíncrona y espera la resolución de la Promise pasada y, a continuación, reanuda la ejecución de la función async y devuelve el valor resuelto.

```
async function getProcessedData(url) {
  let v;
  try {
    v = await downloadData(url);
  } catch(e) {
    v = await downloadFallbackData(url);
  }
  return processDataInWorker(v);
}
```

Observe que, en este ejemplo no hay ninguna instrucción **await** dentro de la instrucción return, porque el valor de retorno de una async function queda implícitamente dentro de un **Promise.resolve**.

La declaración **function*** (la palabra clave function seguida de un asterisco) define una función generadora, que devuelve un objeto **Generator**.
Los generadores son funciones de las que se puede salir y volver a entrar. Su contexto (asociación de variables) será conservado entre las reentradas.

La llamada a una función generadora **no ejecuta su cuerpo inmediatamente**; se devuelve un objeto iterador para la función en su lugar. Cuando el metodo **next()** del iterador es llamado , el cuerpo de la función generadora es ejecutado hasta la primera expresión **yield**, la cual especifica el valor que será retornado por el iterador o con, **yield***, delega a otra función generadora. El método **next()** retorna un objeto con una propiedad value que contiene el valor bajo el operador yield y una propiedad done que indica, con un booleano, si la función generadora ha hecho yield al último valor.

```
function* idMaker(){
  var index = 0;
  while(index < 3)
    yield index++;
}

var gen = idMaker();

console.log(gen.next().value); // 0
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
console.log(gen.next().value); // undefined
// ...
```

Ejemplo con yield*

```
function* anotherGenerator(i) {
  yield i + 1;
  yield i + 2;
  yield i + 3;
}

function* generator(i){
  yield i;
  yield* anotherGenerator(i);
  yield i + 10;
}

var gen = generator(10);

console.log(gen.next().value); // 10
console.log(gen.next().value); // 11
console.log(gen.next().value); // 12
console.log(gen.next().value); // 13
console.log(gen.next().value); // 20
```