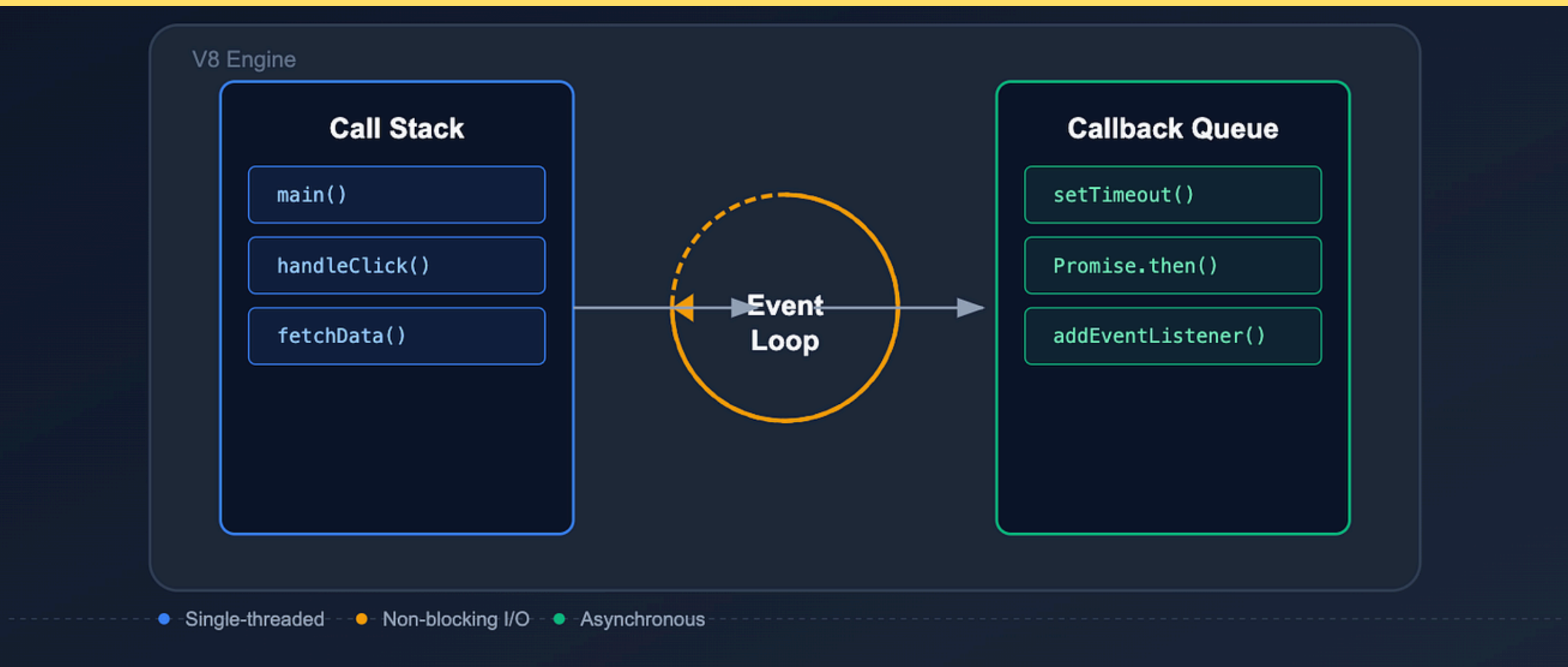
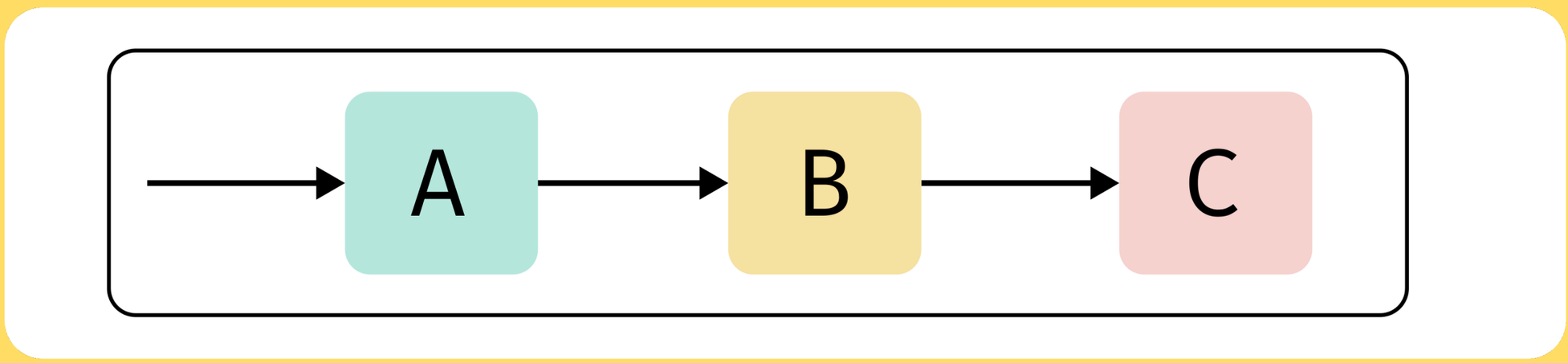
Is javascript a multi-threaded language?



Single-threaded

JavaScript is traditionally a single-threaded language, but thanks to asynchronous functions and Web Workers, it can simulate multi-threaded behavior. 🚀

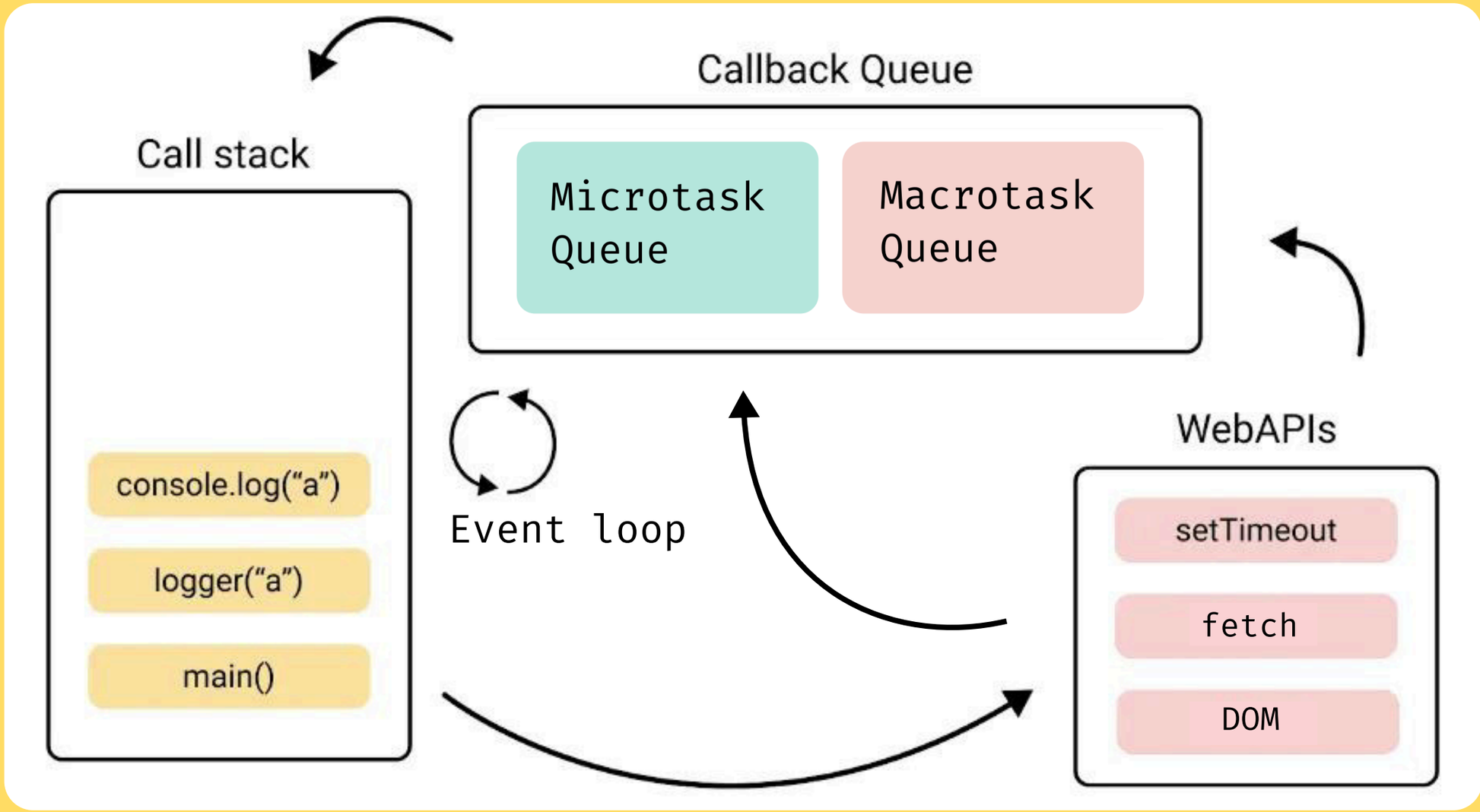


What is asynchronous JavaScript?

Asynchronous JavaScript allows tasks like network requests or timers to run in the background, letting the main thread continue other work. It uses the Event Loop to schedule execution when the main thread is free.

The Event loop

The Event Loop ensures that our asynchronous code runs in the correct order.



Execution Order

1. Synchronous Code
2. Microtasks (Promise callbacks)
3. Macrotasks (setTimeout)

```
// Macro task: setTimeout
setTimeout(() => {
  console.log("Macro task (setTimeout)");
}, 0);

// Micro task: Promise
Promise.resolve().then(() => {
  console.log("Micro task (Promise)");
});
```

Limitations of Asynchronous JavaScript

Dependency on the Event Loop:

JavaScript runs on a single thread, so heavy computations can block the main thread and delay asynchronous tasks.

Asynchronous operations rely on the Event Loop, which might cause delays under heavy load.

Web Workers solve the main thread blocking issue.

Web Workers run JavaScript in a separate thread, allowing heavy tasks without blocking the main thread, keeping the UI responsive.

How does it work?

1. Creation:

A Web Worker is created using the Worker constructor and a separate JavaScript file for the worker script.

```
const worker = new Worker('worker.js');
```

How does it work?

2. Communication:

The main thread and the Web Worker communicate through the postMessage method to send messages and the onmessage event to receive them.

Communication

Main Thread

```
const worker = new Worker('worker.js');

// Send a message to the worker
worker.postMessage('Hello, Worker!');

// Listen for messages from the worker
worker.onmessage = function(event) {
    console.log('Message from worker:', event.data);
};
```

Worker

```
onmessage = function(event) {
    console.log('Message from main thread:', event.data);

    // Send a response back to the main thread
    postMessage('Hello, Main Thread!');
};
```

How does it work?

3. Execution:

The worker runs independently of the main thread, executing tasks like computations or data processing. It doesn't access DOM directly but can use APIs like XMLHttpRequest or fetch.

Execution

Main Thread

```
const worker = new Worker('worker.js');
worker.postMessage(10); // Send a number to calculate factorial

worker.onmessage = function(event) {
    console.log('Factorial result:', event.data);
};
```

Worker

```
onmessage = function(event) {
    const number = event.data;
    const result = factorial(number);
    postMessage(result); // Send the result back to the main thread
};

function factorial(n) {
    return n === 0 ? 1 : n * factorial(n - 1);
}
```

How does it work?

4. Termination:

Workers can be terminated explicitly using worker.terminate() from the main thread. Workers also stop automatically when their script completes.

Termination

Main Thread

```
const worker = new Worker('worker.js');

// Terminate the worker after 5 seconds
setTimeout(() => {
    worker.terminate();
    console.log('Worker terminated.');
}, 5000);
```

Worker

```
setInterval(() => {
    console.log('Worker is still running...');
}, 1000);
```

How can Web Workers help?

1.Prevent UI Freezes: Run heavy tasks in the background, keeping the interface responsive.

Improve Performance: Offload computations to a separate thread.

Real-Time Data: Handle live updates or streaming efficiently.

Asynchronous Tasks: Process large data or complex operations without blocking.

Repeated Tasks: Manage recurring tasks without overloading the main thread.