DEVELOPER REFERENCE

The JavaScript Event Loop

A Developer Reference for Async Execution & Task Scheduling

Version 1.0 | November 2025
Prepared by Avalynn Circe

What This Document Covers

This reference explains how JavaScript's event loop processes synchronous code, microtasks, and macrotasks. Understanding the execution order isn't academic — it directly affects how your async code behaves in production. By the end of this document, you should be able to predict execution order in non-trivial scenarios and reason confidently about race conditions, callback timing, and promise resolution.

The Mental Model

JavaScript is single-threaded. It runs one piece of code at a time. The event loop is the mechanism that decides what runs next. It watches three structures:

- The Call Stack — where currently executing code lives. Functions are pushed when called and popped when they return.
- The Microtask Queue — where resolved Promises and queueMicrotask() callbacks wait. Drained completely before the event loop moves on.
- The Macrotask Queue (Task Queue) — where setTimeout, setInterval, I/O callbacks, and UI events wait. Processes one task per event loop turn.

The execution order is always:

1. Run the current synchronous code to completion (empty the call stack).
2. Drain the microtask queue completely (including any new microtasks added during draining).
3. Run one macrotask from the task queue.
4. Drain the microtask queue again.
5. Repeat from step 3.

> *This order is defined by the HTML specification for browsers and by the Node.js event loop documentation for server environments. The core model is the same; the specific task sources differ.*

Execution Order: A Worked Example

Read the following code and try to predict the output before looking at the explanation below.

```
console.log('A');

setTimeout(() => console.log('B'), 0);

Promise.resolve()
  .then(() => console.log('C'))
  .then(() => console.log('D'));

console.log('E');
```

Output

```
A
E
C
D
B
```

Why

6. A and E are synchronous. They run first, in order, as the call stack processes the script.
7. setTimeout with a delay of 0 does not mean 'run immediately.' It schedules a macrotask. It waits in the task queue.
8. Promise.resolve() creates an already-resolved promise. The .then() callbacks are scheduled as microtasks immediately.
9. After the synchronous code finishes (call stack empty), the event loop drains the microtask queue. C runs, which schedules D as a new microtask. D runs.
10. Only after the microtask queue is empty does the event loop pick up the next macrotask. B runs.

> *The delay argument to setTimeout is a minimum wait, not a guaranteed wait. Even setTimeout(fn, 0) will always yield to pending microtasks before running.*

Promises and Microtasks

Every .then(), .catch(), and .finally() callback is scheduled as a microtask when the promise it is attached to settles. This means promise chains run to exhaustion before any macrotask gets a turn.

```
Promise.resolve()
  .then(() => {
    console.log('microtask 1');
    return Promise.resolve();  // Schedules another microtask
  })
  .then(() => console.log('microtask 2'));
```

```
setTimeout(() => console.log('macrotask'), 0);

// Output:
// microtask 1
// microtask 2
// macrotask
```

When a .then() handler returns a Promise, the next .then() in the chain waits for that promise to settle — but the waiting itself is implemented via microtasks, not macrotasks. The chain stays in the microtask queue.

async/await Under the Hood

async/await is syntactic sugar over Promises. An await expression suspends the async function and schedules the continuation as a microtask once the awaited promise settles. This makes async functions predictable: everything after an await runs in the microtask queue.

```
async function run() {
  console.log('async start');    // synchronous, runs immediately
  await Promise.resolve();       // suspends; schedules continuation
  console.log('after await');    // runs as microtask
}

run();
console.log('synchronous');

// Output:
// async start
// synchronous
// after await
```

The code up to the first await runs synchronously as part of the calling context. The await causes the function to yield, allowing the rest of the synchronous code to run. The continuation resumes as a microtask.

Common Bugs and Their Causes

Bug: Promise Chain That Never Resolves

```
// BAD: returning a new Promise inside .then() without resolving it
fetch('/api/data')
  .then(response => {
    new Promise(resolve => {  // resolve is never called
      // ... forgot to call resolve()
    });
  })
  .then(data => console.log(data));  // never runs
```

If a .then() handler returns a Promise, the next handler waits for it to settle. If the returned Promise never settles, the chain hangs silently. Always ensure the Promise constructor's resolve is called.

Bug: Mixing async/await with .then() Incorrectly

```
// BAD: not awaiting an async function
async function fetchUser(id) {
  const response = await fetch('/users/' + id);
  return response.json();
}

// This does not wait for fetchUser to complete
const user = fetchUser(42);  // user is a Promise, not the resolved value
console.log(user.name);      // undefined

// CORRECT: await the async function
const user = await fetchUser(42);
console.log(user.name);      // works
```

Quick Reference

Code	Status	Description
`setTimeout(fn, 0)`	Macrotask	Runs after all microtasks. Never before a pending .then() callback.
`setInterval(fn, n)`	Macrotask	Same queue as setTimeout. Subject to drift under load.
`Promise.then/catch`	Microtask	Drains completely before next macrotask.
`queueMicrotask(fn)`	Microtask	Explicit microtask scheduling. Same queue as Promise callbacks.
`async/await`	Microtask	Continuation after await is a microtask. Pre-await code is synchronous.
`I/O callbacks`	Macrotask	File reads, network responses. Node.js-specific ordering applies.
`requestAnimationFrame`	Special	Browser only. Fires before paint, after macrotasks.

Further Reading

- Jake Archibald, Tasks, microtasks, queues and schedules (2015) — the canonical visual explainer for browser event loop behavior.
- Node.js Event Loop documentation — covers the libuv phases unique to Node (timers, poll, check, close callbacks).

- MDN Web Docs: Using Promises — comprehensive reference for Promise API and chaining patterns.
- JavaScript: The Parts, Chapter 8 — Asynchronous JavaScript. Covers Promises, async/await, and event loop mechanics with extended worked examples.