

# Day 6 — Event Loop || (Rendering & Pitfalls)

Covered topics:

- Frames & rendering
  - Long tasks & jank
  - Microtask floods & `queueMicrotask`
  - Ordering with `fetch`, `then`, and `await`
  - Debounce & throttle
  - Chunking work
  - Using `requestAnimationFrame`
  - Short performance report (before/after)
- 

## 2. Frames & Rendering

Modern UIs aim for **60 FPS** (frames per second), which means the browser has about **16.67ms** to:

- Run JavaScript
- Handle user input
- Apply style & layout
- Paint and composite the frame

If JavaScript blocks the main thread for too long, the browser **misses frames**, causing visible stutter or "jank".

### Example: Smooth vs. Blocked Rendering

```
// Bad: heavy loop blocks the main thread
button.addEventListener("click", () => {
  const start = performance.now();
  while (performance.now() - start < 1000) {
    // Block for ~1 second
  }
  console.log("Done heavy work");
});
```

Clicking the button will freeze the UI for about 1 second.

---

## 3. Long Tasks & Jank

A **long task** is any piece of JavaScript that runs for a long time on the main thread (e.g., > 50ms). During a long task:

- The browser cannot render new frames.
- User input may be delayed.
- The page feels laggy or frozen.

### Example of a Long Task

```
function longTask() {
  const start = performance.now();
  while (performance.now() - start < 200) {
    // Simulate 200ms of blocking work
  }
}

console.log("Before long task");
longTask();
console.log("After long task");
```

This blocks the main thread for \~200ms, which is more than 10 frames at 60 FPS.

#### Impact:

- Scroll and clicks will feel unresponsive during the task.
- CSS animations may freeze.

## 4. Microtask Floods & `queueMicrotask`

### 4.1 Microtasks Recap

**Microtasks** (e.g., `Promise.then`, `queueMicrotask`) run **after the current call stack** finishes but **before** the browser processes rendering and tasks like `setTimeout`.

If you schedule too many microtasks, you can starve the browser and prevent it from rendering.

### 4.2 Microtask Flood Example (Bad)

```
function flood() {
  queueMicrotask(flood);
}
```

```
flood();
```

This continuously queues new microtasks without giving the browser a chance to:

- Render a frame
- Handle user input
- Process task queue callbacks

**Result:** the page appears frozen.

#### 4.3 When to Use `queueMicrotask`

Use `queueMicrotask` for:

- Small follow-up logic after the current function
- Updating state just after current synchronous code

Do **not** use it inside tight loops or recursive calls that never yield.

---

## 5. Ordering with `fetch`, `then`, and `await`

Network operations (`fetch`) are asynchronous. When they finish, their callbacks (`.then`, `await`) are usually scheduled as **microtasks**.

#### Example: Mixed Sync, Promise, and Timeout

```
console.log("start");

setTimeout(() => {
  console.log("timeout");
}, 0);

Promise.resolve()
  .then(() => console.log("promise 1"))
  .then(() => console.log("promise 2"));

console.log("end");
```

**Execution order:**

1. `start` (synchronous)
2. `end` (synchronous)
3. `promise 1` (microtask)

4. promise 2 (microtask)
5. timeout (task)

This shows that **all microtasks run before any tasks**, even `setTimeout(0)`.

### Example: `fetch` and Promises

```
console.log("start");

fetch("/api/data")
  .then(() => console.log("fetch then"));

Promise.resolve().then(() => console.log("promise"));

console.log("end");
```

Typical order:

1. start
2. end
3. promise
4. fetch then (after the network response is ready)

Both `.then` handlers run as microtasks, but the `fetch` one depends on the network completing.

## 6. Debounce & Throttle Patterns

### 6.1 Debounce

**Debounce** delays executing a function until a certain time has passed since the last call.  
Use it when you care about the **final** action after the user stops typing or resizing.

Common use cases:

- Search input (wait until the user stops typing)
- Window resize handler

#### Implementation:

```
function debounce(fn, delay) {
  let timer;
  return (...args) => {
    clearTimeout(timer);
    timer = setTimeout(() => fn(...args), delay);
```

```

    };
}

const handleSearch = debounce((value) => {
  console.log("Searching for:", value);
}, 300);

input.addEventListener("input", (e) => {
  handleSearch(e.target.value);
});

```

If the user types quickly, the function runs only once after they pause typing.

---

## 6.2 Throttle

**Throttle** ensures a function runs at most once per time interval.  
Use it when you want regular, limited updates during a burst of events.

Common use cases:

- Scroll events
- Drag and mousemove events

**Implementation:**

```

function throttle(fn, limit) {
  let waiting = false;
  return (...args) => {
    if (!waiting) {
      fn(...args);
      waiting = true;
      setTimeout(() => {
        waiting = false;
      }, limit);
    }
  };
}

const handleScroll = throttle(() => {
  console.log("Scroll event");
}, 200);

window.addEventListener("scroll", handleScroll);

```

Even if the scroll event fires many times per second, `handleScroll` runs at most once every 200ms.

---

## 7. Chunking Heavy Work

**chunking** means splitting heavy, long-running work into smaller pieces so the browser can:

- Render between chunks
- Stay responsive to user input

### 7.1 Bad: Single Long Loop

```
function processAll(items) {
  for (let i = 0; i < items.length; i++) {
    // Heavy computation
  }
}
```

If `items` is very large, this loop may block the main thread.

### 7.2 Better: Process in Chunks with `setTimeout`

```
function processInChunks(items, chunkSize = 1000) {
  let index = 0;

  function processChunk() {
    const end = Math.min(index + chunkSize, items.length);

    for (let i = index; i < end; i++) {
      // Heavy computation per item
    }

    index = end;

    if (index < items.length) {
      // Yield back to the event loop and continue later
      setTimeout(processChunk, 0);
    } else {
      console.log("Processing complete");
    }
  }

  processChunk();
}
```

### 7.3 Even Better: Use `requestAnimationFrame` for UI-Related Work

```
function processInAnimationFrames(items, chunkSize = 1000) {
  let index = 0;

  function frame() {
    const end = Math.min(index + chunkSize, items.length);

    for (let i = index; i < end; i++) {
      // Heavy computation per item
    }

    index = end;

    if (index < items.length) {
      requestAnimationFrame(frame);
    } else {
      console.log("Processing complete");
    }
  }

  requestAnimationFrame(frame);
}
```

Using `requestAnimationFrame` aligns work with the browser's rendering cycle.

## 8. Using `requestAnimationFrame` for Smooth UI

`requestAnimationFrame` (rAF) schedules a callback right **before** the next paint.  
It is ideal for:

- Animations
- Smooth UI updates tied to frames

### Example: Smooth Movement

```
const box = document.getElementById("box");
let x = 0;

function move() {
  x += 2;
  box.style.transform = `translateX(${x}px)`;
  requestAnimationFrame(move);
}
```

```
requestAnimationFrame(move);
```

The browser decides the exact timing, making the animation match the refresh rate.

---

## 9. Short Performance Report

### 9.1 Before (Problem)

- **Symptoms:** UI freezes and feels laggy when processing large datasets or handling many user events.
- **Findings:**
  - A single long loop was running for several hundred milliseconds.
  - Scroll and input handlers were firing on every event.
  - Some logic used Promises in a way that created many microtasks in a row.

### 9.2 Changes Applied

#### 1. Chunk heavy work

2. Replaced a single large loop with chunked processing using `setTimeout` / `requestAnimationFrame`.
3. This gave the browser chances to render between chunks.

#### 4. Debounced input handlers

5. Wrapped search input handler with `debounce(fn, 300)`.
6. Reduced unnecessary network calls and CPU usage.

#### 7. Throttled scroll listeners

8. Wrapped scroll handler with `throttle(fn, 200)`.
9. Lowered the frequency of expensive operations during scrolling.

#### 10. Avoided microtask floods

11. Removed recursive `Promise.then` patterns that scheduled many microtasks without yielding.
12. Ensured that long-running work yields back to the event loop.

### **9.3 After (Result)**

- The UI remains responsive during heavy operations.
- Scrolling and typing feel smoother, with fewer dropped frames.
- CPU usage is more stable, and long tasks are either eliminated or significantly reduced.
- Rendering can occur regularly, allowing the browser to maintain close to 60 FPS in typical scenarios.