Understanding the event loop

Here's a **detailed explanation** of the **JavaScript Event Loop**, one of the most fundamental concepts in asynchronous programming:

🧠 What is the Event Loop?

The **event loop** is a mechanism that allows JavaScript (which is single-threaded) to handle **asynchronous operations** like I/O, timers, and network requests **without blocking** the main execution thread.

> It lets JavaScript do one thing at a time, but still handle multiple tasks via callbacks, promises, etc.

🏗️ Components Involved

1. **Call Stack**
   - A stack data structure where functions are pushed when called and popped when finished.
   - JS executes code **synchronously** on this stack.
2. **Web APIs (Browser APIs)**
   - Provided by the browser (not JavaScript itself) — e.g., `setTimeout`, `DOM events`, `fetch`.
   - When you call them, they run in the background.
3. **Callback/Task Queue (Macrotask Queue)**
   - Stores callbacks from Web APIs to be executed **after the current call stack is empty**.
4. **Microtask Queue**
   - Stores resolved **Promises** and `queueMicrotask()` callbacks.
   - These are executed **before** anything in the callback queue.

5. **Event Loop**
    - Continuously checks:
        1. Is the **call stack** empty?
        2. If yes, does the **microtask queue** have anything? Run all microtasks.
        3. Then, move to the **macrotask queue** and push one task onto the stack.

🔁 How It Works (Step-by-step)

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

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

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

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

🔷 Execution Flow:

1. `"Start"` → printed immediately (call stack)
2. `setTimeout()` → offloaded to Web APIs → callback goes to **macrotask queue**
3. `Promise.then()` → goes to **microtask queue**
4. `"End"` → printed immediately
5. Call stack is now empty
6. Event loop processes **microtasks** → `"Promise"`
7. Then processes **macrotask** → `"Timeout"`

Output:

```
Start
End
Promise
Timeout
```

🧪 Microtasks vs Macrotasks

Queue Type	Examples	Priority
Microtasks	`Promise.then()`, `async/await`, `queueMicrotask()`	🔺 High
Macrotasks	`setTimeout`, `setInterval`, `setImmediate` (Node.js)	🔻 Lower

Microtasks always execute **before** the next macrotask starts.

🛠️ Use Cases

- Avoid blocking UI in browser apps.
- Perform heavy operations in non-blocking chunks using async APIs.
- Chain async steps (e.g., `fetch → parse → render`).

🧠 Summary

- JavaScript is **single-threaded**, but handles async tasks using **event loop**, **task queues**, and **Web APIs**.
- **Microtasks** (like promises) are prioritized over **macrotasks** (like timers).
- Understanding this helps you avoid bugs like delayed rendering, unhandled promises, and race conditions.

Let me know if you want a visual diagram of the event loop in action or an animation-based resource!