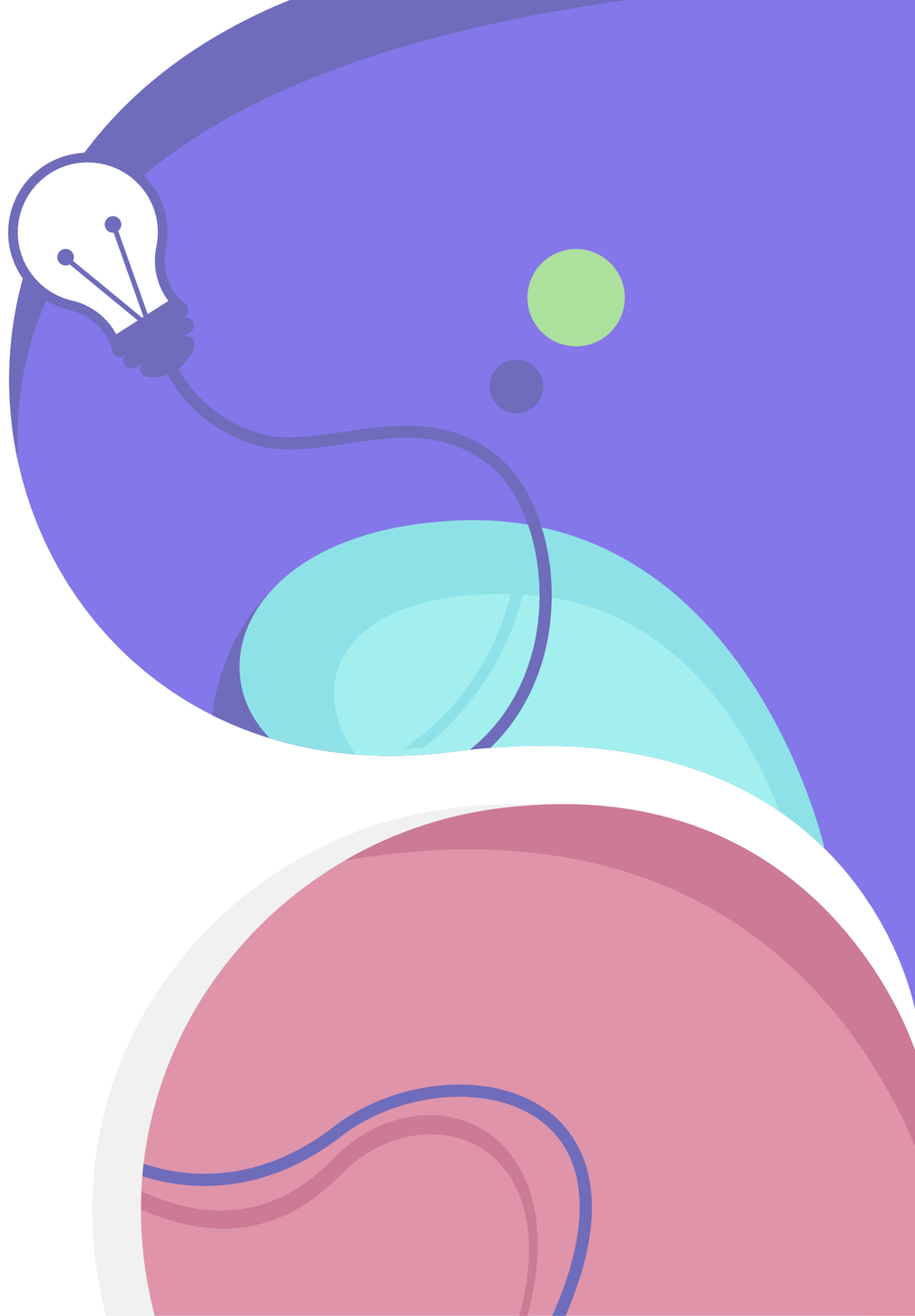
JavaScript Master từ A-Z
Một vài slides bài giảng lý thuyết



Trungquandev

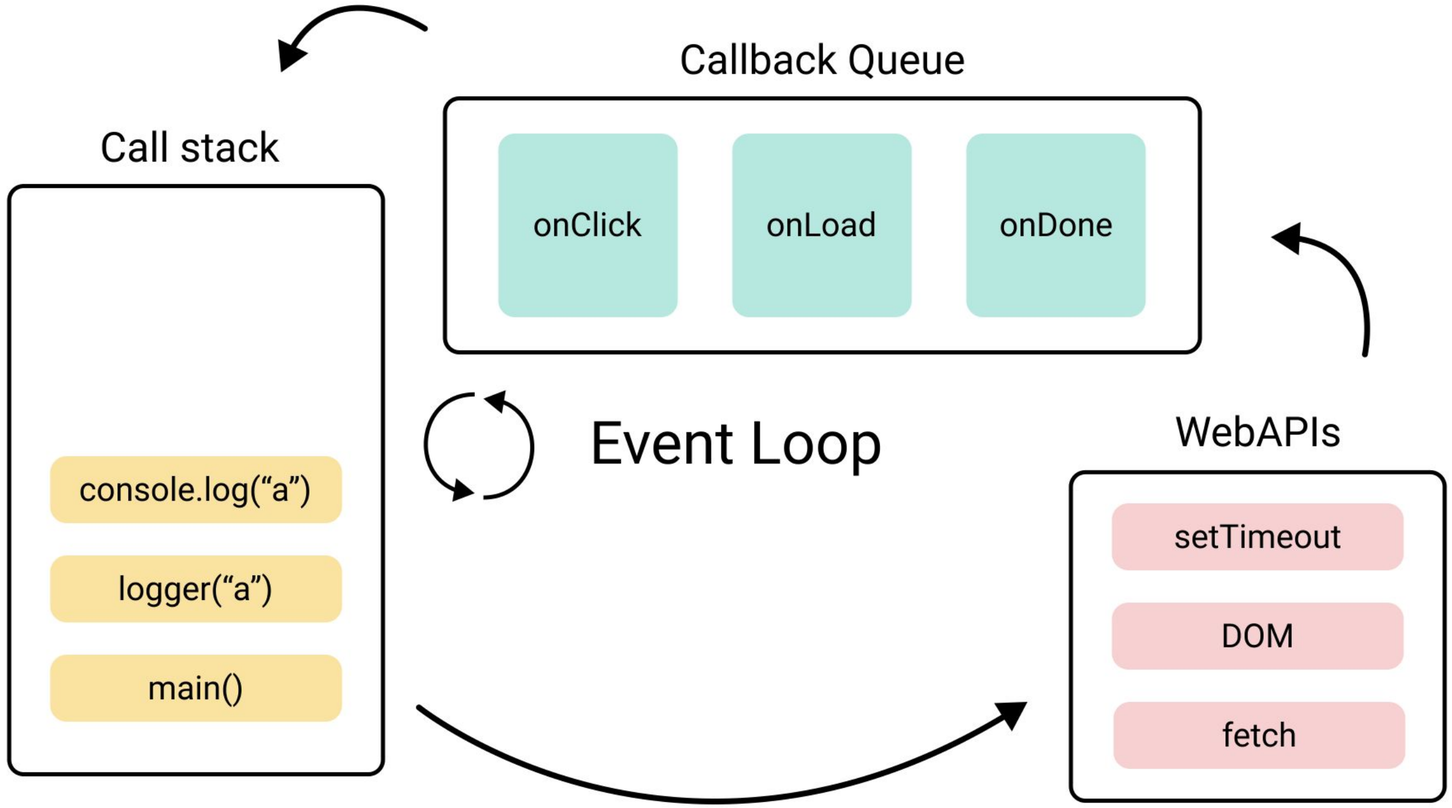
Trung Quân Dev

https://trungquandev.com

Event Loop: Câu hỏi phỏng vấn huyền thoại

- **JavaScript** là **ngôn ngữ đơn luồng (single-threaded)**, nghĩa là nó chỉ có thể thực hiện **một công việc tại một thời điểm**.

- Bởi vậy nếu như không có **cơ chế bất đồng bộ** (**Async**) hoạt động dựa trên **Event Loop** (“động cơ” chạy phía sau) thì câu chuyện **Block Luồng** sẽ xảy ra.



Event Loop huyền thoại trong JS

Event Loop là một **cơ chế** giúp JavaScript quản lý cách mà các tác vụ **bất đồng bộ** được xử lý và thực thi theo đúng thứ tự.

Hình dung thực tế:

Bất đồng bộ - Async: giống như việc **bạn gọi món ăn tại nhà hàng**, rồi tiếp tục **làm việc khác** trong lúc chờ.

Event Loop: giống như **người phục vụ**, kiểm tra khi món ăn đã sẵn sàng và mang đến cho bạn.

Ví dụ phổ biến thường sử dụng trên trang web:

- **Gửi request đến server:** `fetch`, `XMLHttpRequest`
- **Đợi một khoảng thời gian** mới thực thi code: `setTimeout`, `setInterval`

Event Loop: 4 th ành phần chính

1. **Call Stack (Ngăn xếp thực thi)** – Nơi chứa các hàm đang được thực thi.

2. **Web APIs** – Các API của trình duyệt giúp xử lý những tác vụ bất đồng bộ (như `setTimeout`, `fetch, DOM Event`).

3. **Queue (Hàng đợi):** Nơi chứa các hàm callback chờ được thực thi. **Bao gồm 2 loại:**

Callback Queue (chứa `setTimeout`, `fetch, DOM Event`) **và Microtask Queue** (chứa `Promise, MutationObserver, queueMicrotask`)

- Lưu ý: **Microtask Queue** chứa các tác vụ có độ ưu tiên hơn **Callback Queue**
- Ví dụ: **Promise** đưa vào **Microtask Queue** sẽ ưu tiên chạy trước **setTimeout** trong **Callback Queue**

4. **Event Loop** – Cơ chế như một vòng lặp liên tục giúp kiểm tra và di chuyển các hàm từ **Callback Queue** hoặc **Microtask Queue** vào **Call Stack** để thực thi => không bị block luồng đơn của JavaScript.

Event Loop: Quy tr ình hoạt động

1. JavaScript thực thi code trong **Call Stack** theo thứ tự đồng bộ và tuần tự.
2. Nếu gặp một tác vụ bất đồng bộ (`setTimeout`, `fetch`, `DOM event`), **nó không được thực thi ngay lập tức mà** được gửi đến **Web APIs** để xử lý.
3. Khi tác vụ bất đồng bộ hoàn thành, **callback** của nó sẽ được đưa vào **Callback Queue** hoặc **Microtask Queue tùy loại.**
4. **Event Loop** lúc này như một vòng lặp liên tục quan sát **Call Stack** và **Queue (Callback, Microtask)**

- **Khi Call Stack rỗng**, Event Loop sẽ kiểm tra **Microtask Queue** trước rồi đến **Callback Queue** xem có chứa các Tác vụ hay các hàm Callback nào không.
- **Nếu có**: hàm Callback đầu tiên sẽ được lấy ra từ **Queue** và đưa vào **Call Stack** để thực thi.
- **Nếu không có gì trong Queue**: Event Loop sẽ tiếp tục đợi cho đến khi có Tác vụ / hàm Callback mới được thêm vào.

Event Loop: Ví dụ quy tr ình hoạt động

Step 1: `console.log("Start")` chạy ngay lập tức (đồng bộ).
Step 2: `setTimeout(...)` được gửi đến **Web API** để hẹn giờ 2 giây (nó không chặn chương trình).
Step 3: `console.log("End")` chạy ngay lập tức.
Step 4: Sau 2 giây, callback từ `setTimeout` được đưa vào **Callback Queue**.
Step 5: **Event Loop kiểm tra Call Stack:**

- Nếu Call Stack rỗng, nó đưa callback từ **Callback Queue** vào Call Stack để thực thi: `console.log("Async Task")`

```
1  console.log("Start")
2
3  setTimeout(() => {
4    console.log("Async Task")
5  }, 2000)
6
7  console.log("End")
```

Event Loop: Thứ tự ưu tiên của Queue

Kết quả in ra lần lượt như sau:

1. `console.log("Start")` chạy đầu tiên (đồng bộ).
2. `console.log("End")` chạy tiếp theo (đồng bộ).
3. `console.log("Promise resolve")` chạy tiếp theo, vì nằm trong Microtask Queue ưu tiên cao.
4. `console.log("queueMicrotask done")` chạy tiếp theo, cũng nằm trong Microtask Queue ưu tiên cao, nhưng code nằm sau promise.
5. `console.log("setTimeout done")` chạy cuối cùng, dù để 0 giây nhưng nó nằm trong Callback Queue độ ưu tiên thấp hơn Microtask Queue

```
1  console.log("Start")
2
3  setTimeout(() => {
4    console.log("setTimeout done")
5  }, 0)
6
7  Promise.resolve().then(() => {
8    console.log("Promise resolved")
9  })
10
11 queueMicrotask(() => {
12   console.log("queueMicrotask done")
13 })
14
15 console.log("End")
```

Cấu trúc cơ bản của một trang web

Một trang web cơ bản gồm 3 thành phần chính:

- **HTML** (HyperText Markup Language): Xây dựng cấu trúc trang web.
- **CSS** (Cascading Style Sheets): Định dạng, tạo kiểu cho trang web.
- **JavaScript**: Tạo các tương tác động trên trang, **thao tác với DOM.**

Cơ bản cách mà trình duyệt render một trang web như thế nào?

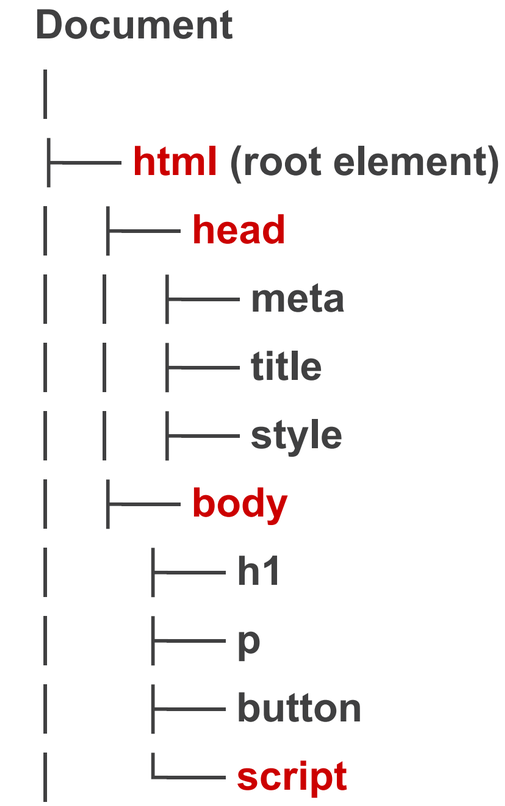
Khi một trang web được tải, trình duyệt sẽ thực hiện các bước sau:

- Đọc (parse) mã HTML để tạo ra cây DOM.
- Đọc và áp dụng CSS để tạo ra cây CSSOM (CSS Object Model).
- Kết hợp DOM và CSSOM để tạo ra trang hoàn chỉnh.
- Sử dụng JavaScript để thao tác với DOM (truy cập, thay đổi, hoặc xóa các phần tử trên trang)

DOM (Document Object Model)

- **DOM là một tiêu chuẩn Mô hình Đối tượng Tài liệu được W3C (World Wide Web Consortium) đưa ra. Mục đích để cho JavaScript có thể thao tác với 2 dạng tài liệu:**
  - **HTML** (Ngôn ngữ đánh dấu được thiết kế để hiển thị nội dung trên trình duyệt.)
  - **XML** (ngôn ngữ đánh dấu dùng để lưu trữ và truyền dữ liệu theo cấu trúc.)
- **Chúng ta sẽ tập trung vào HTML DOM:**
- **Khi trình duyệt tải một trang web, nó sẽ phân tích (parse) HTML và tạo ra một mô hình cây DOM, giúp chúng ta có thể truy cập, thay đổi, hoặc xóa các phần tử trên trang.**

Cấu trúc của cây HTML DOM



Ngoài ra có cái ảnh trông khá cũ trên W3School:
https://www.w3schools.com/js/js_htmldom.asp

—

DOM được tổ chức theo cấu trúc cây. Mỗi thẻ HTML là một Node.

Các loại node trong DOM:

- **Document Node**: Đại diện cho toàn bộ tài liệu HTML
- **Element Node**: Đại diện cho thẻ HTML (`<h1>`, `<p>`, `<div>`...vv)
- **Attribute Node**: Đại diện cho thuộc tính của phần tử (`id`, `class`, `href`,`src`...vv).
- **Text Node**: Đại diện cho nội dung bên trong một thẻ HTML