Event Loop (Rendering & Pitfalls)

▼ Frames

Nowadays, the user expects the web page to be more interactive and respond quickly to his actions.

The refresh rate of a display is an important consideration when we talk about building a website that focuses on user experience. ⇒ **by refresh here we mean updating the screen based on an action**

Commonly, browsers refresh their screens 60 times per second, and each refresh produces the visual output that the user sees on the screen **(frame)**.

A frame is the visual update that the browser produces on the screen, so by achieving the refresh rate of 60 frames per second, each frame in this case will be rendered in approximately 16.6 ms (in reality only 10 ms due to the browser overhead for each frame, which refers to the time the browser itself needs to do internal work to render a frame, not including JavaScript).

So the browser needs to do all the work related to every single frame in only 10 ms, and this work includes this pipeline:

1. Running related JS code
2. Calculating CSS styles ⇒ Figuring out which CSS rules apply to which HTML elements
3. Computing layout (reflows) ⇒ Here the browser determines how much space the element will take up from the whole page, and where it should be placed
4. Painting pixels ⇒ This process includes filling in the determined pixels by drawing the elements and apply their styles and colors on them in multiple layers
5. Compositing layers into the final image ⇒ This happens by applying the created layers to the screen in the correct order.

If the browser takes more than 10 ms to do the render a frame, then some frames are skipped and the layout not responsive enough, and this phenomenon is called **jank.**

▼ Long Tasks

They are any piece of JS code the keeps running in the call stack for more than 50ms, like parsing heavy script or doing heavy DOM manipulations which create costly web page reflows.

They are bad for performance and user experience, as the browser cannot process or handle any other action during this time (the screen freezes).

In order to solve this problem, we can break down long tasks into smaller ones using **setTimeout**, which will postpone the execution of its handler into another task, even if we set the time to 0.

We can detect long tasks using the browser API **PerformanceObserver.**

▼ Patterns used to avoid jank / UI freeze

1. **Debounce**

   It is used to ensure that the function will run only once after the user stops doing something.

   For example, when the user searches for something, normally the function that handles this event will be executed each time the user change the search term.

   That's why we use the debounce pattern in order to delay the execution of that handler sometime after the user stops updating the search term and then executes the handler only one time.

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

```
}

const search = debounce(() => {
  console.log("Searching API...");
}, 300);
```

2. **Throttling**

   It allows the function to run once every specific time, even if it is triggered many times.

   This can be very useful in many cases like scroll, drag, mouse move, where the event handler runs only once after some time, decreasing the number of times it is get executed in

```
function throttle(fn, delay) {
  let waiting = false;
  return function (...args) {
    if (waiting) return;
    waiting = true;
    fn.apply(this, args);
    setTimeout(() => waiting = false, delay);
  };
}

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

Microtask Flood

By default, promise handlers (.then() / .catch()) are pushed inside the microtask queue to be executed later in their completion order when the call stack becomes empty.

Also, tasks inside the microtask queue has higher priority than tasks inside the macrotask queue, where a single task from the macrotask queue gets executed after the execution of the all of the tasks inside the microtask queue.

In addition to that, we can create another microtask during the execution of other microtask using the queueMicrotask() API, which can lead to the addition of a lot of tasks inside the microtask queue, so that it will not become empty for a long time, which prevents the processing of any other task or rendering a new frame, and this case is called **microtask flood.**

We can solve this problem by using **requestAnimationFrame** API or **setTimeout** to schedule tasks later and yield back to the browser in order to process or handle another task, so that the UI does not freeze.

Patterns used to chunk tasks

1. **setTimeout**

```
function heavyWork() {
  let i = 0;

  function chunk() {
    for (let j = 0; j < 500000; j++) {
      i++;
    }
    if (i < 5e7) setTimeout(chunk);
  }

  chunk();
```

```
/*
we want to run 50,000,000 operations, but we need to split it into smaller
tasks (500,000 operations per task) in order not to freeze the UI, so the web
page can respond to other actions or render the next frame
*/
```

2. **requestAnimationFrame**

   It runs right before the rendering of each frame, allowing adding some updates to each rendering.

```
function heavyWork() {
  let i = 0;

  function chunk() {
    for (let j = 0; j < 300000; j++) {
      i++;
    }
    if (i < 5e7) requestAnimationFrame(chunk);
    /*
    requestAnimationFrame schedule to run chunk fn before the next frame,
    allowing the browser to render the current frame
    */
  }

  chunk();
}

  /*
  we want to run 50,000,000 operations, but we need to split it into smaller
  chucnks (300,000 operations per task), so the browser can render each frame
  in its corresponding 10ms without any frame skipping
  */
```

▼ queueMicrotask API

Unlike setTimeout which adds the function to the macrotask queue, it allows a function to be handled as a microtask.

This is useful when we want a function to get executed as soon as possible after the execution of all of the microtasks finishes before any rendering or handling of any events.

```
console.log("script start");
const i = 0;
queueMicrotask(() => console.log("microtask ran"));
const promise = new Promise((resolve, reject)=>{if(i == 0) resolve(5)})
promise.then((data) => console.log(data))
console.log("script end");
/*
1. script start
2. script end
3. microtask run
4. 5
*/
```

▼ Ordering with fetch, .then & await

1. **fetch**

- When you call fetch the browser starts the network request outside the JS engine (the networking layer / Web API).
- fetch returns a promise immediately. That Promise will be **settled later** when the network response arrives (or rejected on network error).
- When the promise settles, any .then() / .catch() reactions scheduled for that promise are queued as **microtasks**.

- **.then()**
  - A .then() callback is a microtask.
  - Microtasks run after the current call stack finishes, and before the browser renders the next frame and before macrotasks.
  - If microtasks enqueue more microtasks, the engine keeps running microtasks until the microtask queue is empty, so microtasks can delay rendering if too many are queued.
- **await**

  await can only appear inside an async function. When the engine reaches await:
  - The async function pauses (it yields) and returns control to the event loop. The async function itself returns a promise.
  - The rest of the async function (the code *after* the await) is scheduled as a microtask (like .then() on the awaited promise) to be run after other sync code & all other queued microtasks.
  - await does not block the whole script or the main thread, it only pauses that async function.

    Other synchronous code, microtasks already queued, rendering, and macrotasks may proceed according to the usual event loop rules.

▼ Resources

https://web.dev/articles/rendering-performance#:~:text=Image%3A A user interacting with,feel responsive to user input

https://web.dev/articles/optimize-long-tasks#:~:text=The browser blocks interactions from,very long periods of time