Gmail Bikash Adhikari <virtual.bikash@gmail.com>

what is the event loop in JavaScript ? What is it's role in asynchronous in JavaScript? explain with example

1 message

Bikash <bikash.caim@gmail.com> Fri, Mar 7, 2025 at 6:56 PM
To: virtual.bikash@gmail.com

Understanding the JavaScript event loop is crucial for grasping how asynchronous operations work in this single-threaded language. Here's a breakdown:

What is the Event Loop?

- **Single-Threaded Nature:**
  - JavaScript operates on a single thread, meaning it executes one task at a time. This could lead to blocking if long-running operations were executed synchronously.
- **The Event Loop's Role:**
  - The event loop is a mechanism that allows JavaScript to handle asynchronous operations efficiently. It continuously monitors the call stack and the task queue (or callback queue).
  - Its job is to check if the call stack is empty. If it is, it takes the first task from the task queue and pushes it onto the call stack for execution.

Role in Asynchronous JavaScript

- **Non-Blocking Behavior:**
  - The event loop enables non-blocking behavior. When an asynchronous operation (like `setTimeout`, `fetch`, or a promise) is encountered, it's handed off to the browser's Web APIs.
  - The main thread continues executing the rest of the code, and when the asynchronous operation completes, its callback function is placed in the task queue.
- **Managing Callbacks:**
  - The event loop ensures that these callback functions are executed in the correct order, after the main thread has finished its current tasks.
- **Microtasks and Macrotasks:**
  - The event loop works with two types of queues:
    - **Microtask Queue:** Holds microtasks like promise callbacks. Microtasks are processed with a higher priority.
    - **Macrotask Queue:** Holds macrotasks like `setTimeout` callbacks and user events.

- The event loop will empty the microtask queue before it goes to the macrotask queue.

Example

JavaScript

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

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

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

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

Explanation:

1. `console.log("Start")` executes and "Start" is printed.
2. `setTimeout` is called. The callback is placed in the macrotask queue.
3. `Promise.resolve().then()` is called. The promise callback is placed in the microtask queue.
4. `console.log("End")` executes and "End" is printed.
5. The call stack is now empty. The event loop checks the microtask queue.
6. The promise callback is moved to the call stack and executed, so "Promise callback" is printed.
7. The microtask queue is empty. The event loop checks the macrotask queue.
8. The `setTimeout` callback is moved to the call stack and executed, so "Timeout callback" is printed.

Output:

```
Start
End
Promise callback
Timeout callback
```

In essence, the event loop is the engine that drives JavaScript's ability to handle asynchronous operations without freezing the user interface.

Sources
1. https://juejin.cn/post/7377326783582076968