

[Khaled Abdelrahman][[Link](#)]

Team

First, let's talk about eventloop in javascript:

An event loop is often times the “main” of a program that handles events. If you have written a program you know that it will exit when it is done. Programs that run indefinitely like UI applications, video games, web servers, etc will have some form of an event loop. They can come in different flavors such as being a polling loop (check if there is an event every time it loops) or it can be handled via blocks and something triggers it to come and do it's job.

For instance, imagine you have a program running a UI. If it's idle and you look at your running processes it's likely not going to be eating up a ton of processing time by continually looping. It can either be utterly silent or it may be taking up a fraction of a percent of the CPUs time. But if you go and hit a button the program is going to wake up and do something. This is ran by an event loop. Clicking that button fired an event which got processed by the event loop which called the buttons onClick event handler.

What about eventloop in Node.js?

Node.js is a single-threaded event-driven platform that is capable of running non-blocking, asynchronously programming. These functionalities of Node.js make it memory efficient. The event loop allows Node.js to perform non-blocking I/O operations despite the fact that JavaScript is single-threaded. It is done by assigning operations to the operating system whenever and wherever possible. Most operating systems are multi-threaded and hence can handle multiple operations executing in the background. When one of these operations is completed, the kernel tells Node.js and the respective callback assigned to that operation is added to the event queue which will eventually be executed. This will be explained further in detail later in this topic.

Features of Event Loop:

- 1- Event loop is an endless loop, which waits for tasks, executes them and then sleeps until it receives more tasks.
- 2- The event loop executes tasks from the event queue only when the call stack is empty i.e. there is no ongoing task.

3-The event loop allows us to use callbacks and promises. The event loop executes the tasks starting from the oldest first.

```
console.log("This is the first statement");

setTimeout(function() {
    console.log("This is the second statement");
}, 1000);

console.log("This is the third statement");
```

Output

This is the first statement

This is the third statement

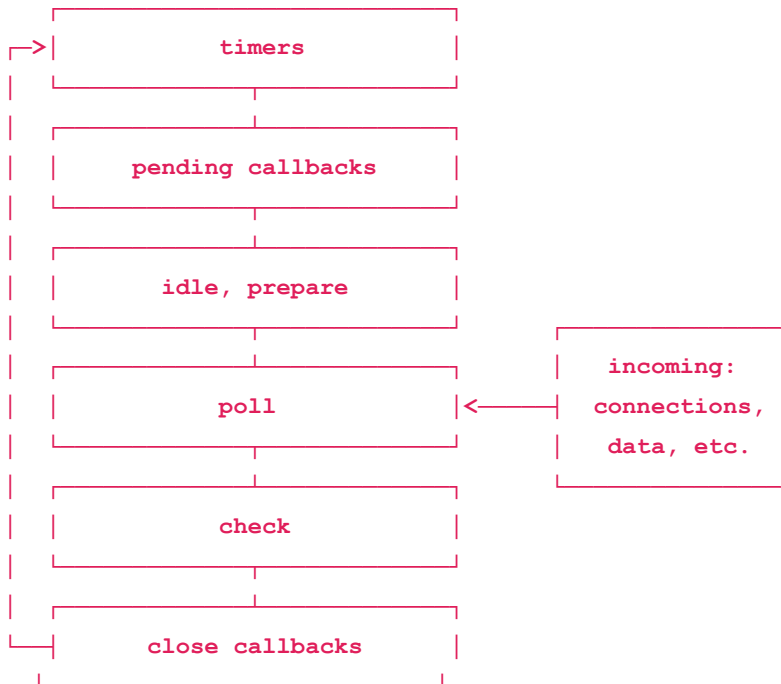
then,

This is the second statement

In the above example, the first console log statement is pushed to the call stack and "This is the first statement" is logged on the console and the task is popped from the stack. Next, the setTimeout is pushed to the queue and the task is sent to the Operating system and the timer is set for the task. This task is then popped from the stack. Next, the third console log statement is pushed to the call stack and "This is the third statement" is logged on the console and the task is popped from the stack. When the timer set by setTimeout function (in this case 1000 ms) runs out, the callback is sent to the event queue. The event loop on finding the call stack empty takes the task at the top of the event queue and sends it to the call stack. The callback function for setTimeout function runs the instruction and "This is the second statement" is logged on the console and the task is popped from the stack.

Resources: [youtube](#), [Quora](#), [geeksforgeeks](#)

[Ahmed M.Osman] [\[Link\]](#)
Individually



3:18

Each box will be referred to as a "phase" of the event loop

3:18

timers: this phase executes callbacks scheduled by `setTimeout()` and `setInterval()`.

pending callbacks: executes I/O callbacks deferred to the next loop iteration.

idle, prepare: only used internally.

poll: retrieve new I/O events; execute I/O related callbacks (almost all with the exception of close callbacks, the ones scheduled by timers, and `setImmediate()`); node will block here when appropriate.

check: `setImmediate()` callbacks are invoked here.

close callbacks: some close callbacks, e.g. `socket.on('close', ...)`.

Between each run of the event loop, Node.js checks if it is waiting for any asynchronous I/O or timers and shuts down cleanly if there are not any.

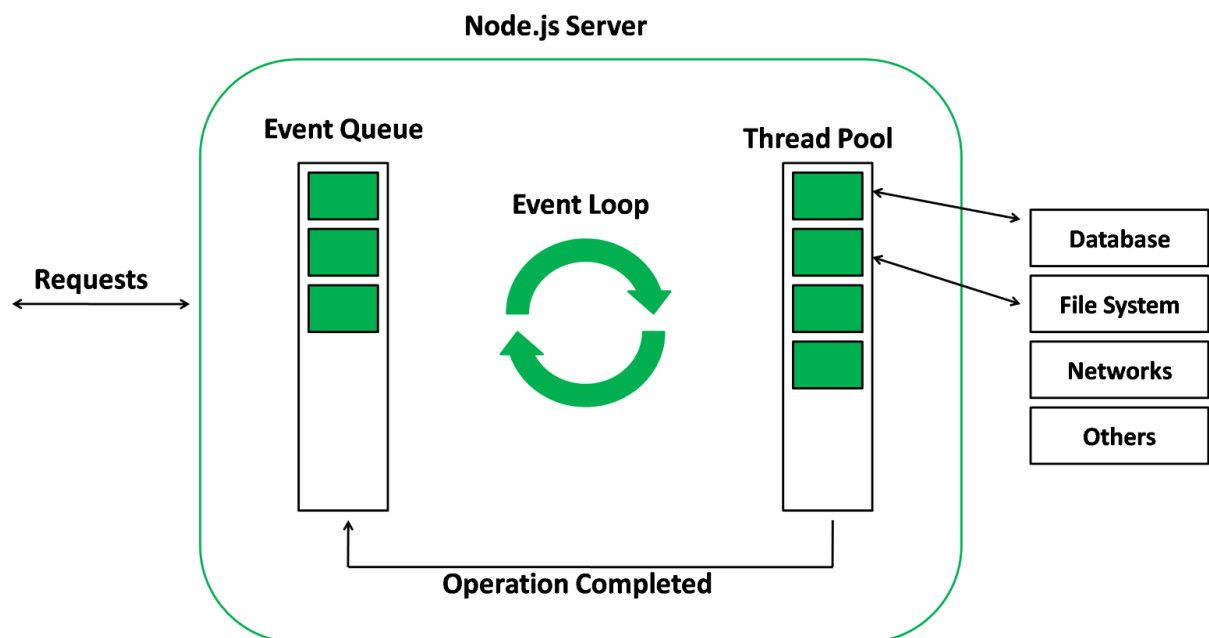
**[Team] (Ibrahim Hafez, Alaa Magdi,
Mohamed Adel)**

Event Loop

What is the Event Loop?

The event loop is what allows Node.js to perform non-blocking I/O operations — despite the fact that JavaScript is single-threaded — by offloading operations to the system kernel whenever possible.

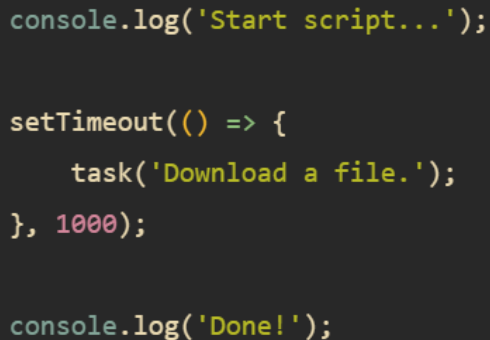
Since most modern kernels are multi-threaded, they can handle multiple operations executing in the background. When one of these operations completes, the kernel tells Node.js so that the appropriate callback may be added to the poll queue to eventually be executed. We'll explain this in further detail later in this topic.



Another example in Java Script

```
console.log('Start script...');

setTimeout(() => {
    task('Download a file.');
```



```
}, 1000);

console.log('Done!');
```

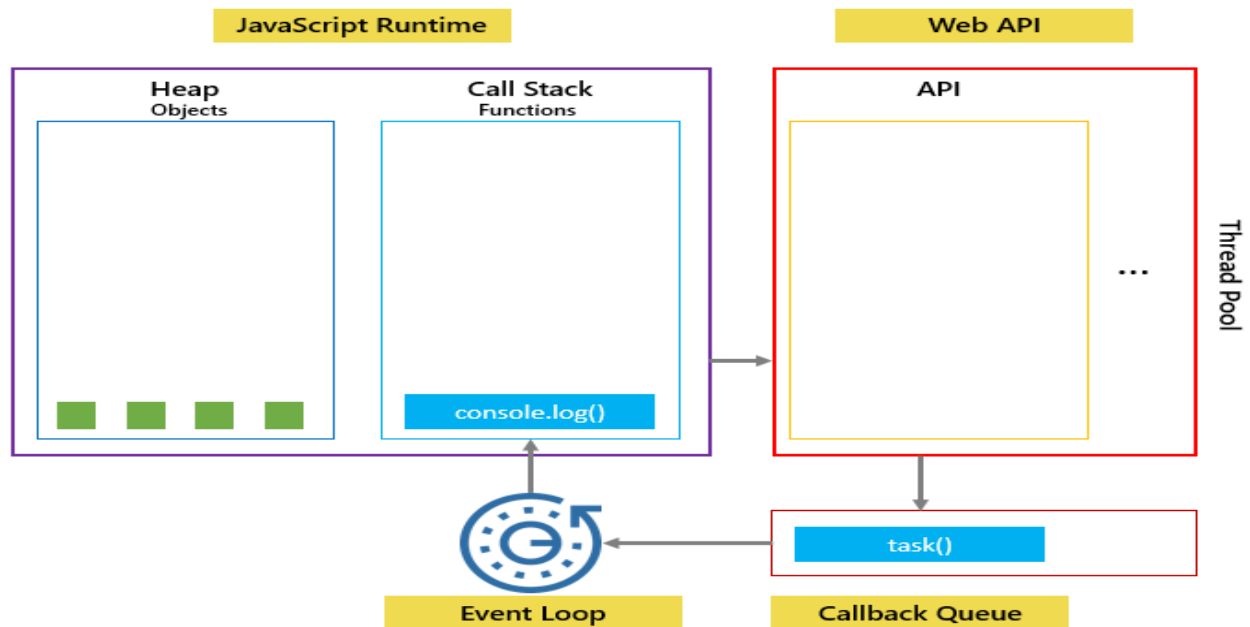
In this example, you'll see the message 'Start script...' and 'Done!' immediately. And after that, you'll see the message 'Download a file'.

So, JavaScript runtime uses a message queue, which is a list of messages to be processed. Each message has an associated function that gets called to handle the message.

At some point during the event loop, the runtime starts handling the messages on the queue, starting with the oldest one. To do so, the message is removed from the queue and its corresponding function is called with the message as an input parameter. As always, calling a function creates a new stack frame for that function's use.

The processing of functions continues until the stack is once again empty. Then, the event loop will process the next message in the queue (if there is one).

Check the figure below:



So, that means that there is Another detail important to mention, is that Node.js expects to return quickly to the client. So, if a callback function works with a very long computation, the single-thread will be busy with it, which can stop the event-loop completely which makes this a disadvantage to use the Event loop among all the good advantages of doing so.