**EVENT LOOP**

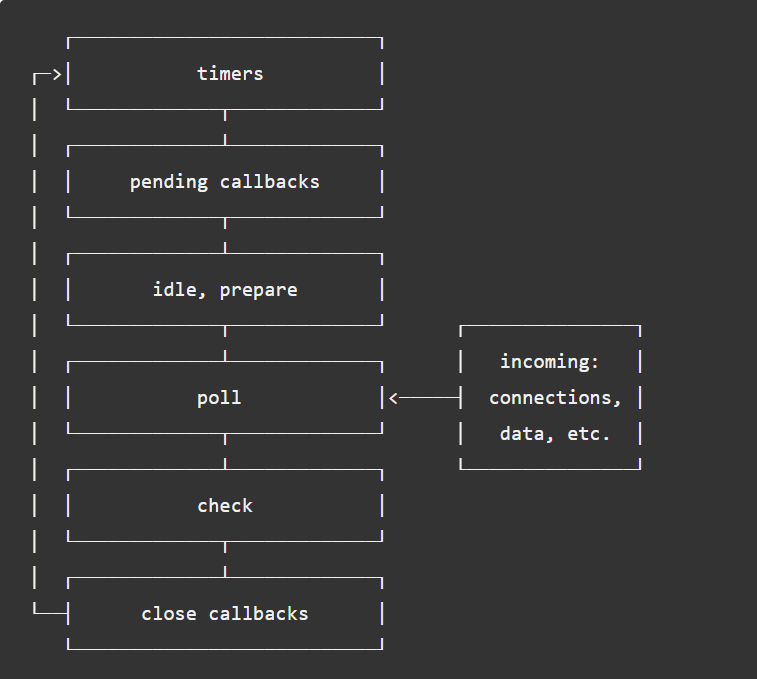
Event loop:

* 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.
* 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.

Features of Event Loop:

* Event loop is an endless loop, which waits for tasks, executes them and then sleeps until it receives more tasks.
* The event loop executes tasks from the event queue only when the call stack is empty i.e. there is no ongoing task.
* The event loop allows us to use callbacks and promises.
* The event loop executes the tasks starting from the oldest first.

Working of Event Loop:



* When Node.js starts, it initializes the event loop, processes the provided input script (or drops into the REPL, which is not covered in this document) which may make async API calls, schedule timers, or call process.nextTick(), then begins processing the event loop.
* Each phase has a FIFO queue of callbacks to execute. While each phase is special in its own way, generally, when the event loop enters a given phase, it will perform any operations specific to that phase, then execute callbacks in that phase's queue until the queue has been exhausted or the maximum number of callbacks has executed. When the queue has been exhausted or the callback limit is reached, the event loop will move to the next phase, and so on.
* Since any of these operations may schedule *more* operations and new events processed in the **poll** phase are queued by the kernel, poll events can be queued while polling events are being processed. As a result, long running callbacks can allow the poll phase to run much longer than a timer's threshold.

Phases overview:

* 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', ...).

Timers:

* A timer specifies the **threshold** *after which* a provided callback *may be executed* rather than the **exact** time a person *wants it to be executed*. Timers callbacks will run as early as they can be scheduled after the specified amount of time has passed; however, Operating System scheduling or the running of other callbacks may delay them.
* Technically, the [**poll** phase](https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/) controls when timers are executed.

const fs = require('fs');

function someAsyncOperation(callback) {

// Assume this takes 95ms to complete

fs.readFile('/path/to/file', callback);

}

const timeoutScheduled = Date.now();

setTimeout(() => {

const delay = Date.now() - timeoutScheduled;

console.log(`${delay}ms have passed since I was scheduled`);

}, 100);

// do someAsyncOperation which takes 95 ms to complete

someAsyncOperation(() => {

const startCallback = Date.now();

// do something that will take 10ms...

while (Date.now() - startCallback < 10) {

// do nothing

}

});

Pending Callbacks:

* This phase executes callbacks for some system operations such as types of TCP errors. For example if a TCP socket receives ECONNREFUSED when attempting to connect, some \*nix systems want to wait to report the error. This will be queued to execute in the pending callbacks phase.

Poll:

* The poll phase has two main functions:
* Calculating how long it should block and poll for I/O, then
* Processing events in the poll queue.
* When the event loop enters the poll phase and there are no timers scheduled, one of two things will happen:
* If the poll queue is not empty, the event loop will iterate through its queue of callbacks executing them synchronously until either the queue has been exhausted, or the system-dependent hard limit is reached.
* If the poll queue is empty, one of two more things will happen:
* If scripts have been scheduled by setImmediate(), the event loop will end the poll phase and continue to the check phase to execute those scheduled scripts.
* If scripts have not been scheduled by setImmediate(), the event loop will wait for callbacks to be added to the queue, then execute them immediately.
* Once the poll queue is empty the event loop will check for timers whose time thresholds have been reached. If one or more timers are ready, the event loop will wrap back to the timers phase to execute those timers' callbacks.

Check:

* This phase allows a person to execute callbacks immediately after the poll phase has completed. If the poll phase becomes idle and scripts have been queued with setImmediate(), the event loop may continue to the check phase rather than waiting.
* setImmediate() is actually a special timer that runs in a separate phase of the event loop. It uses a libuv API that schedules callbacks to execute after the poll phase has completed.
* Generally, as the code is executed, the event loop will eventually hit the poll phase where it will wait for an incoming connection, request, etc. However, if a callback has been scheduled with setImmediate() and the poll phase becomes idle, it will end and continue to the check phase rather than waiting for poll events.

Close callback:

* If a socket or handle is closed abruptly (e.g. socket.destroy()), the 'close' event will be emitted in this phase. Otherwise it will be emitted via process.nextTick().

**setImmediate() vs setTimeout():**

* setImmediate() and setTimeout() are similar, but behave in different ways depending on when they are called.
* setImmediate() is designed to execute a script once the current poll phase completes.
* setTimeout() schedules a script to be run after a minimum threshold in ms has elapsed.
* The order in which the timers are executed will vary depending on the context in which they are called. If both are called from within the main module, then timing will be bound by the performance of the process (which can be impacted by other applications running on the machine).

// timeout\_vs\_immediate.js

setTimeout(() => {

console.log('timeout');

}, 0);

setImmediate(() => {

console.log('immediate');

});

* The main advantage to using setImmediate() over setTimeout() is setImmediate() will always be executed before any timers if scheduled within an I/O cycle, independently of how many timers are present.

Process.nextTick():

* Every time the event loop takes a full trip, we call it a tick.
* When we pass a function to process.nextTick(), we instruct the engine to invoke this function at the end of the current operation, before the next event loop tick starts.

WHY?

1. Allow users to handle errors, cleanup any then unneeded resources, or perhaps try the request again before the event loop continues.
2. At times it's necessary to allow a callback to run after the call stack has unwound but before the event loop continues.