**The Node.js Event Loop**

**Event Loop**

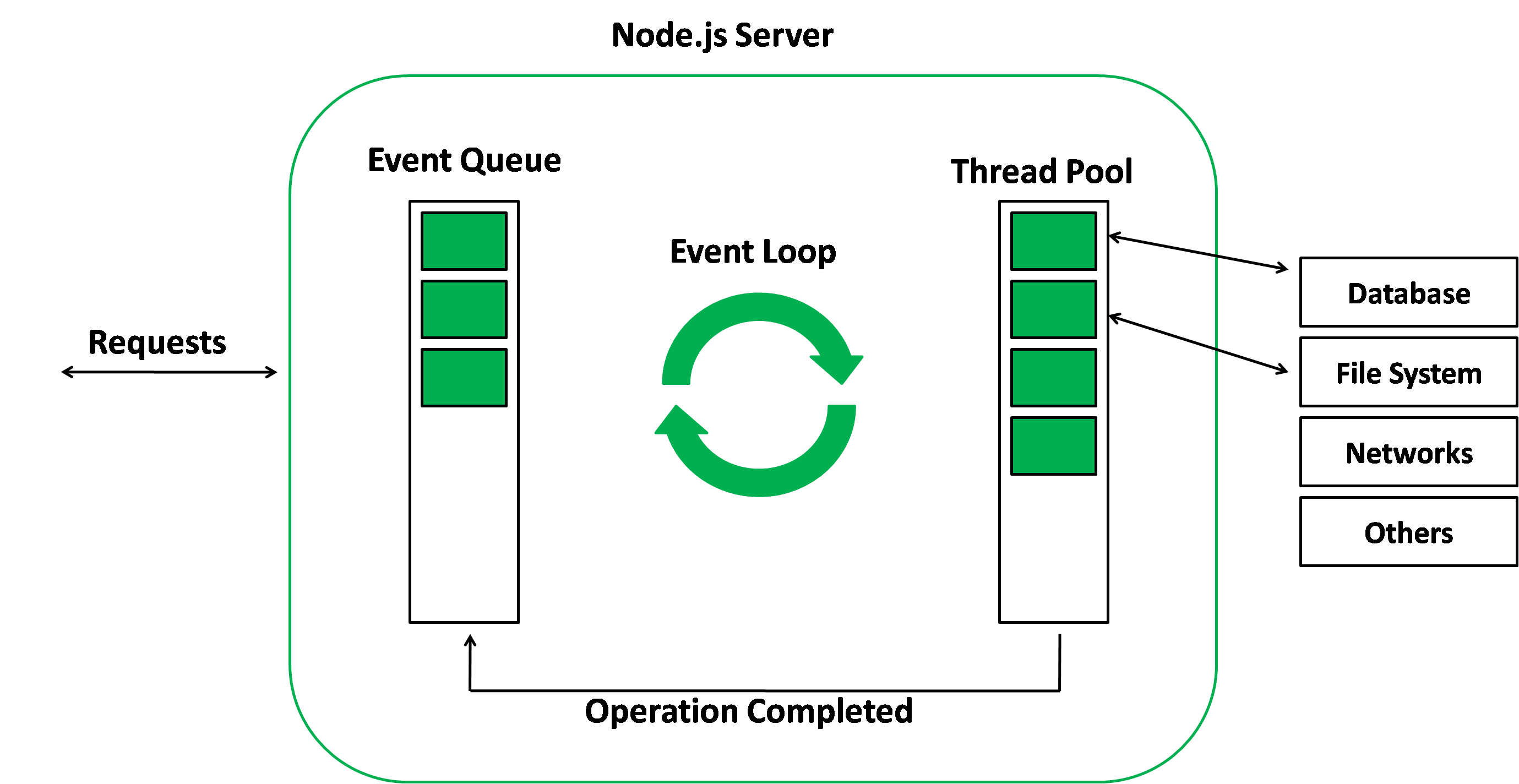
* Event loop is an endless loop, **which waits for tasks, executes them and then sleeps until it receives more tasks**. The event loop executes tasks from the event queue only when the call stack is empty i.e. there is no ongoing task.
* The **event loop** allows Node.js to perform non-blocking I/O operations despite the fact that JavaScript is single-threaded. It is done by assigning operations to the operating system whenever and wherever possible.

**The call stack**

* The call stack is a LIFO (Last In, First Out) stack.
* The event loop continuously checks the **call stack** to see if there's any function that needs to run.
* While doing so, it adds any function call it finds to the call stack and executes each one in order.

**Libuv**

* When using Node.js, a special library module called libuv is used to perform async operations. This library is also used, together with the back logic of Node, to manage a special thread pool called the libuv thread pool.This thread pool is composed of four threads used to delegate operations that are too heavy for the event loop.



**Microtasks and Macrotasks**

To be more accurate there are actually two types of queues:  
1. The macro-task queue (or just called the task queue).  
2. The micro-task queue.

The event loop should process the micro-task queue entirely, after processing one macro-task from the macro-task queue.

Examples of microtasks - process.nextTick , promises

Examples of macrotasks – setTimeout, setInterval, setImmediate

**Macrotasks:**

**setTimeout()**

* Execute a specified block of code once after a specified time has elapsed.

**setInterval()**

* Execute a specified block of code repeatedly with a fixed time delay between each call.

**setImmediate()**

* The setImmediate function is **used to execute a function right after the current event loop finishes**.
* It is the same as calling the setTimeout function with zero delays.

**Microtasks:**

**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.**

Calling setTimeout(() => {}, 0) will execute the function at the end of next tick, much later than when using nextTick() which prioritizes the call and executes it just before the beginning of the next tick.

Use nextTick() when you want to make sure that in the next event loop iteration that code is already executed.

**Promises**

* It's a way to execute the result of an async function as soon as possible, rather than being put at the end of the call stack.
* Promises that resolve before the current function ends will be executed right after the current function.

**Illustrated examples:**

console.**log**(1);

**setTimeout**(function **foo**() {

    console.**log**('foo');

}, 6500);

**setTimeout**(function **boo**() {

    console.**log**('boo');

}, 2500);

**setTimeout**(function **baz**() {

    console.**log**('baz');

}, 0);

for (const value of ['A','B']) {

    console.**log**(value);

}

function **two**() {

    console.**log**(2);

}

**two**();

//Output

1

2

boo

foo

Explanation:

1. console.log(1) method is called and placed on the call stack and being executed.
2. SetTimeout is being executed, the console.log(‘foo’) is moved to SetTimeout Web Api, and 0 milliseconds afterward it moves to **Macro**-Task Queue.
3. Promise.resolve() is being called, it is being resolved and then .then() method is moved to **Micro**-Task queue.
4. console.log(2) method is called and placed on the call stack and being executed.
5. Event Loop sees that the call-stack is empty, it takes firstly the task from Micro-Task queue which is the Promise task, puts the console.log(‘boo’) on the call-stack and executes it.
6. Event Loop sees that the call-stack is empty, then it sees that the Micro-Task is empty, then it takes the next task from the Macro-Task queue which is the SetTimeout task, puts the console.log(‘foo’) on the call-stack and executes it.

console.**log**(1);

**setTimeout**(function **foo**() {

    console.**log**('foo');

}, 6500);

**setTimeout**(function **boo**() {

    console.**log**('boo');

}, 2500);

**setTimeout**(function **baz**() {

    console.**log**('baz');

}, 0);

for (const value of ['A','B']) {

  console.**log**(value);

}

function **two**() {

    console.**log**(2);

}

**two**();

//Output

1

A

B

2

baz

boo

foo

Explanation:

1. First console.log(1) is printed
2. Then the for loop is executed
3. The two() function is called and hence console.log(2) statement is executed
4. And then the setTimeout() statements are executed according to the timeout time given.

**Microtasks and Macrotasks working:**

Step 1: The event loop updates the loop time to the current time for the current execution.

Step 2: Micro-Queue is executed.

Step 3: A task from the Timers phase is executed.

Step 4: Checking if there is something in the Micro-Queue and executes the whole Micro-Queue if there is something.

Step 5: Returns to Step 3 until the Timers phase is empty.

Step 6: A task from the Pending Callbacks phase is executed.

Step 7: Checking if there is something in the Micro-Queue and executes the whole Micro-Queue if there is something.

Step 8: Returns to Step 6 until the Pending Callbacks phase is empty.

And then Idle… Micro-Queue … Poll … Micro-Queue … Check … Micro-Queue … Close CallBacks and then it starts over.

**Phases of the Event loop:**

So, the previous steps with only the “Callback Queue” and then with “Macro and Micro Queues” were abstract explanations about how the Event Loop works. The phases of the event loop are as follows:

* **Timers:** Callbacks scheduled by setTimeout() or setInterval() are executed in this phase.
* **Pending Callbacks:** I/O callbacks deferred to the next loop iteration are executed here.
* **Idle, Prepare:** Used internally only.
* **Poll:** Retrieves new I/O events.
* **Check:** It invokes setIntermediate() callbacks.
* **Close Callbacks:** It handles some close callbacks. Eg: socket.on(‘close’, …)

