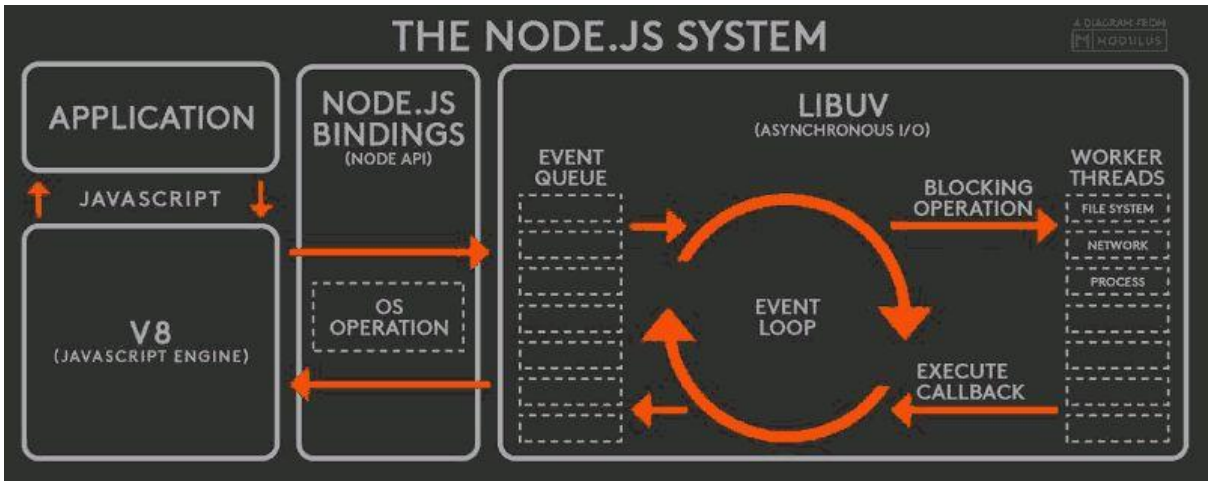
Node.JS Architecture

Basically, Node.js is a combination of Google's V8 JavaScript engine, an event loop, and a low-level I/O API.

Below diagram denotes a simplified version of Node.js architecture.



Following are the 3 main parts

- V8 Engine
- js Bindings (Node API)
- An event loop

JavaScript Engine

A JavaScript engine is a program or an interpreter which executes JavaScript code.

Below mentioned are some of the various JavaScript engines :-

- Rhino from Mozilla
- JavaScriptCode developed by Apple for Safari
- JerryScript a lightweight engine for Internet of things
- Chakra (JScript9) for Internet Explorer
- Chakra (JavaScript) for Microsoft Edge
- V8 from Google and so on.

Since Node.js uses Google V8, coverage of other JavaScript engines are beyond the scope of this article.

Google V8

V8 is Google's open source JavaScript engine, written in C++. It is not only used in Google Chrome but is also the part of popular Node.js. V8 directly translates JavaScript code into efficient machine code using JIT (Just-In-Time) compiler instead of using an interpreter.

Inside, the V8 engine, consists of several threads.

- A thread for fetching JS code, compiling it & then executing it.
- A separate thread for compiling, so that the main thread can keep executing while the former is optimizing the code .
- A Profiler thread which tells the runtime on which methods we spend more time so that Crankshaft can try to optimize them.
- Garbage Collection threads

Full-Codegen – At start , V8 utilizes Full-Codegen compiler which directly transforms parsed JavaScript code into machine code without any intermediate bytecode, this is why V8 does not needs an interpreter. This allows to start execution of the code as soon as possible.

Crankshaft – Now another thread starts using another compiler "Crankshaft" . Its main role is optimization.

In-lining – Inlining replaces the line of code where the function is called with the body of the called function for faster execution. C++ developers are used to with this concept.

Hidden Classes – It's difficult for compiler to optimize dynamically typed language such as JavaScript because types of objects can be changed at runtime. At runtime V8 creates hidden classes that are attached objects to track their layout. So it is very important to maintain the shape of your objects so that V8 is able to optimize code efficiently.

Inline Caching – Inline caching relies on the observation that repeated calls to the same method tend to occur on the same type of object. V8 uses a cache for the type of objects that were passed as a parameter in recent method calls and uses this knowledge to make an assumption about the type of object that will be passed as a parameter in the future. If this assumption is right, it can skip the process of figuring out how to

access the object's properties and can use the stored information from cache .

Garbage collection – Garbage collection is freeing up memory which is not in use anymore. Languages such as C have APIs for this like free() but JavaScript lacks such API. So this memory management task is handled by V8.

Event Loop

Node.js is an event-based platform. This means that everything that happens in Node is the reaction to an event. A transaction passing through Node traverses a cascade of callbacks. Abstracted away from the developer, this is all handled by a library called libuv which provides a mechanism called an event loop.

There is only one thread that executes JavaScript code and this is the thread where the event loop is running. The execution of callbacks (know that every user code in a running Node.js application is a callback) is done by the event loop.

Libuv by default creates a thread pool with four threads to offload asynchronous work to. Today's operating systems already provide asynchronous interfaces for many I/O tasks (e.g. AIO on Linux). Whenever possible, libuv will use those asynchronous interfaces, avoiding usage of the thread pool. The same applies to third party subsystems like databases. Here the authors of the driver will rather use the asynchronous interface than utilizing a thread pool. In short: Only if there is no other way, the thread pool will be used for asynchronous I/O.

While there are queue-like structures involved, the event loop does not run through and process a stack. The event loop as a process is a set of phases with specific tasks that are processed in a round-robin manner.

Node JS – Single Threaded Event Loop

Node JS Platform does not follow Request/Response Multi-Threaded Stateless Model. It follows Single Threaded with Event Loop Model. Node JS Processing model mainly based on Javascript Event based model with Javascript callback mechanism.

As Node JS follows this architecture, it can handle more and more concurrent client requests very easily. Before discussing this model internals, first go through the diagram below.

The main heart of Node JS Processing model is “Event Loop”. If we understand this, then it is very easy to understand the Node JS Internals.

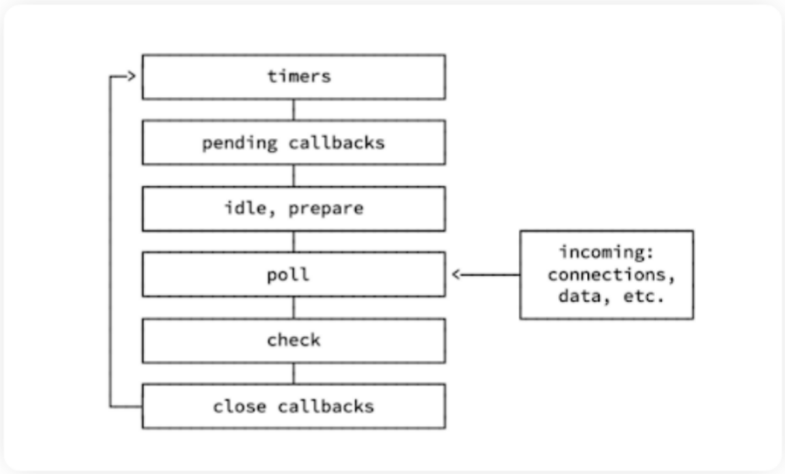
Single Threaded Event Loop Model Processing Steps:

- Clients Send request to Web Server.
- Node JS Web Server internally maintains a Limited Thread pool to provide services to the Client Requests.
- Node JS Web Server receives those requests and places them into a Queue. It is known as “Event Queue”.
- Node JS Web Server internally has a Component, known as “Event Loop”. Why it got this name is that it uses indefinite loop to receive requests and process them.
- Event Loop uses Single Thread only. It is main heart of Node JS Platform Processing Model.
- Even Loop checks any Client Request is placed in Event Queue. If no, then wait for incoming requests for indefinitely.
- If yes, then pick up one Client Request from Event Queue
    - Starts process that Client Request
    - If that Client Request Does Not requires any Blocking IO Operations, then process everything, prepare response and send it back to client.
    - If that Client Request requires some Blocking IO Operations like interacting with Database, File System, External Services then it will follow different approach
        - Checks Threads availability from Internal Thread Pool
        - Picks up one Thread and assign this Client Request to that thread.
        - That Thread is responsible for taking that request, process it, perform Blocking IO operations, prepare response and send it back to the Event Loop
        - Event Loop in turn, sends that Response to the respective Client.

The Event Loop

The event loop consists of the following phases:

- Timers
- Pending callbacks
- Idle/prepare
- Poll----I/o polling
- Check
- Close callbacks,
- Incoming connections and data



The most important phase is the first phase– the timers. Timers are callbacks registered with '**setTimeout()**' or '**setInterval()**'.

They also allow us to monitor the event loop with the option to schedule data, ultimately offering a good way to check if an event is idle. The event loop then executes expired timers and checks for pending callbacks again.

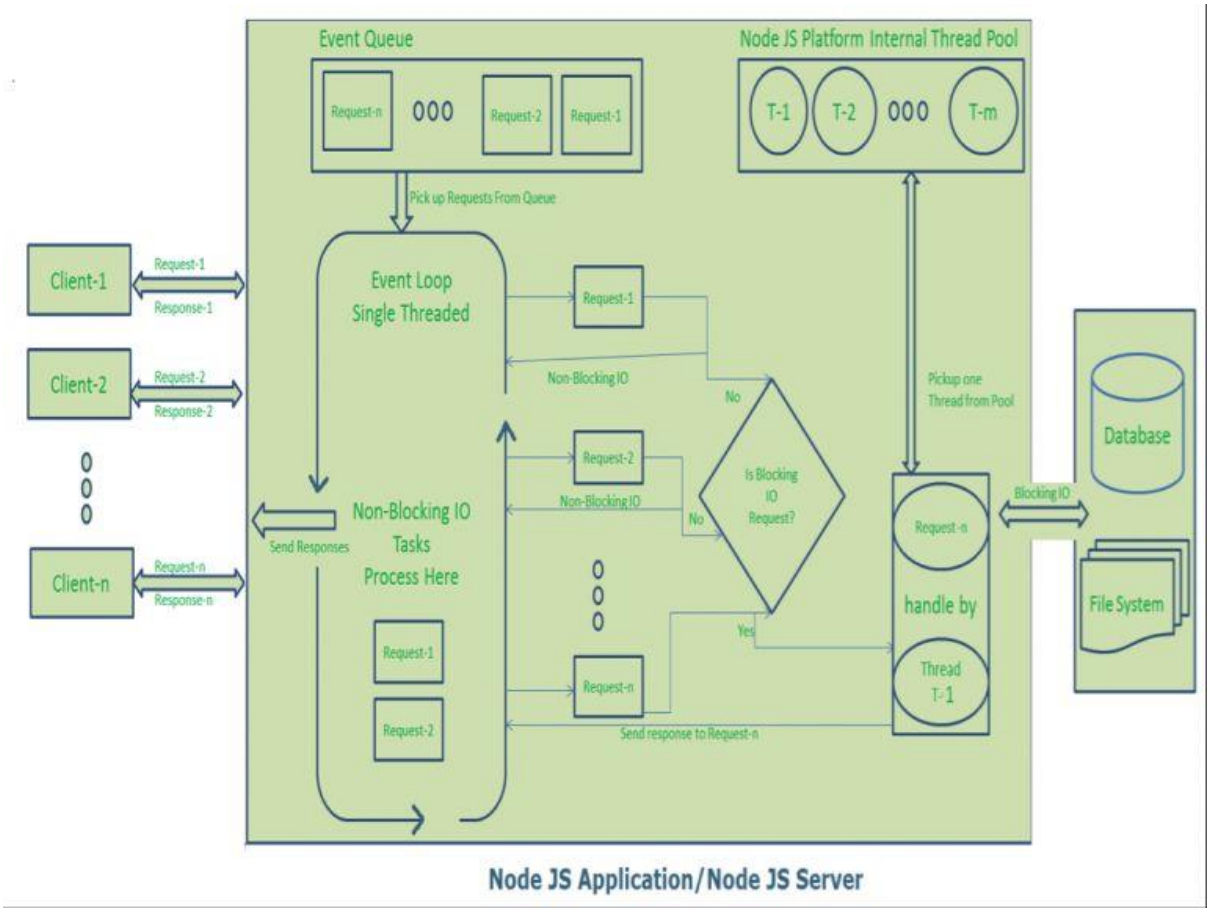
The I/O callbacks are checked first in the poll phase, followed by the '**setImmediate()**' callbacks. Node.js also has a special callback, the **process.nextTick(),** which executes after each loop phase. This callback has the highest priority.

During the poll phase, the event loop looks for events that have completed their asynchronous tasks and are ready to be processed.

We then move to the check phase, during which the event loop executes all the callbacks registered with '**setImmediate()**.

Close callbacks are associated with closing network connections or handling errors during I/O events. The event loop will then look for scheduled timers.

The loop then continues, keeping the application responsive and non-blocking



As in the above diagram, “n” number of clients send request to web server if they are accessing web application concurrently and are Client-1, Client-2… and Client-n. Web server internally maintains a limited thread pool which if assumed as “m” number of threads in thread pool. Node JS Web Server receives Client-1, Client-2… and Client-n Requests and places them in the Event Queue. Node JS Even Loop Picks up those requests one by one.

- Even Loop pickups Client-1 Request-1
    - Checks whether Client-1 Request-1 does require any Blocking IO Operations or takes more time for complex computation tasks.
    - As this request is simple computation and Non-Blocking IO task, it does not require separate Thread to process it.
    - Event Loop process all steps provided in that Client-1 Request-1 Operation (Here Operations means Java Script’s functions) and prepares Response-1
    - Event Loop sends Response-1 to Client-1
- Even Loop pickups Client-2 Request-2

        - Checks whether Client-2 Request-2does require any Blocking IO Operations or takes more time for complex computation tasks.
        - As this request is simple computation and Non-Blocking IO task, it does not require separate Thread to process it.
        - Event Loop process all steps provided in that Client-2 Request-2 Operation and prepares Response-2
        - Event Loop sends Response-2 to Client-2
- Even Loop pickups Client-n Request-n
    - Checks whether Client-n Request-n does require any Blocking IO Operations or takes more time for complex computation tasks.
    - As this request is very complex computation or Blocking IO task, Even Loop does not process this request.
    - Event Loop picks up Thread T-1 from Internal Thread pool and assigns this Client-n Request-n to Thread T-1
    - Thread T-1 reads and process Request-n, perform necessary Blocking IO or Computation task, and finally prepares Response-n
    - Thread T-1 sends this Response-n to Event Loop
    - Event Loop in turn, sends this Response-n to Client-n

Here Client Request is a call to one or more Java Script Functions. Java Script Functions may call other functions or may utilize its Callback functions nature.

Node JS – Single Threaded Event Loop Advantages

- Handling more and more concurrent client's request is very easy.
- Even though our Node JS Application receives more and more Concurrent client requests, there is no need of creating more and more threads, because of Event loop.
- Node JS application uses less Threads so that it can utilize only less resources or memory

Blocking vs Non-Blocking

This overview covers the difference between blocking and non-blocking calls in Node.js. This overview will refer to the event loop and libuv but no

prior knowledge of those topics is required. Readers are assumed to have a basic understanding of the JavaScript language and Node.js callback pattern. “I/O” refers primarily to interaction with the system’s disk and network supported by libuv.

Blocking – Blocking is when the execution of additional JavaScript in the Node.js process must wait until a non-JavaScript operation completes. This happens because the event loop is unable to continue running JavaScript while a blocking operation is occurring.

In Node.js, JavaScript that exhibits poor performance due to being CPU intensive rather than waiting on a non-JavaScript operation, such as I/O, isn’t typically referred to as blocking. Synchronous methods in the Node.js standard library that use libuv are the most commonly used blocking operations. Native modules may also have blocking methods.

All of the I/O methods in the Node.js standard library provide asynchronous versions, which are non-blocking, and accept callback functions. Some methods also have blocking counterparts, which have names that end with Sync.

Comparing Code – Blocking methods execute synchronously and non-blocking methods execute asynchronously. Using the File System module as an example, this is a synchronous file read:

```
const fs = require(‘fs’);

const data = fs.readFileSync(‘/file.md’); // blocks here until file is read
```

And here is an equivalent asynchronous example:

```
const fs = require(‘fs’);

fs.readFile(‘/file.md’, (err, data) => {

if (err) throw err;

});
```

The first example appears simpler than the second but has the disadvantage of the second line blocking the execution of any additional

JavaScript until the entire file is read. Note that in the synchronous version if an error is thrown it will need to be caught or the process will crash. In the asynchronous version, it is up to the author to decide whether an error should throw as shown.

Let's expand our example a little bit:

```
const fs = require('fs');

const data = fs.readFileSync('/file.md'); // blocks here until file is read

console.log(data);

// moreWork(); will run after console.log
```

And here is a similar, but not equivalent asynchronous example:

```
const fs = require('fs');

fs.readFile('/file.md', (err, data) => {

if (err) throw err;

console.log(data);

});

// moreWork(); will run before console.log
```

In the first example above, console.log will be called before moreWork(). In the second example fs.readFile() is non-blocking so JavaScript execution can continue and moreWork() will be called first. The ability to run moreWork() without waiting for the file read to complete is a key design choice that allows for higher throughput.

Concurrency and Throughput – JavaScript execution in Node.js is single threaded, so concurrency refers to the event loop's capacity to execute JavaScript callback functions after completing other work. Any code that is expected to run in a concurrent manner must allow the event loop to continue running as non-JavaScript operations, like I/O, are occurring.

As an example, let's consider a case where each request to a web server takes 50ms to complete and 45ms of that 50ms is database I/O that can be done asynchronously. Choosing non-blocking asynchronous operations frees up that 45ms per request to handle other requests. This is a significant difference in capacity just by choosing to use non-blocking methods instead of blocking methods.

The event loop is different than models in many other languages where additional threads may be created to handle concurrent work.

Dangers of Mixing Blocking and Non-Blocking Code – There are some patterns that should be avoided when dealing with I/O. Let's look at an example:

```
const fs = require('fs');

fs.readFile('/file.md', (err, data) => {

if (err) throw err;

console.log(data);

});

fs.unlinkSync('/file.md');
```

In the above example, fs.unlinkSync() is likely to be run before fs.readFile(), which would delete file.md before it is actually read. A better way to write this, which is completely non-blocking and guaranteed to execute in the correct order is:

```
const fs = require('fs');

fs.readFile('/file.md', (readFileErr, data) => {

if (readFileErr) throw readFileErr;

console.log(data);

fs.unlink('/file.md', (unlinkErr) => {

if (unlinkErr) throw unlinkErr;
```

```
});

});
```

The above places a non-blocking call to fs.unlink() within the callback of fs.readFile() which guarantees the correct order of operations.