Server-Side Programming & Node.js Intro

Front end and Back end

- Front end / Client-side
  - HTML, CSS and Javascript
  - Asynchronous request handling and AJAX

- Back end / Server-side
  - Node.js, PHP, Python, Ruby, Perl
  - Compiled languages like C#, Java or Go
  - Various technologies and approaches

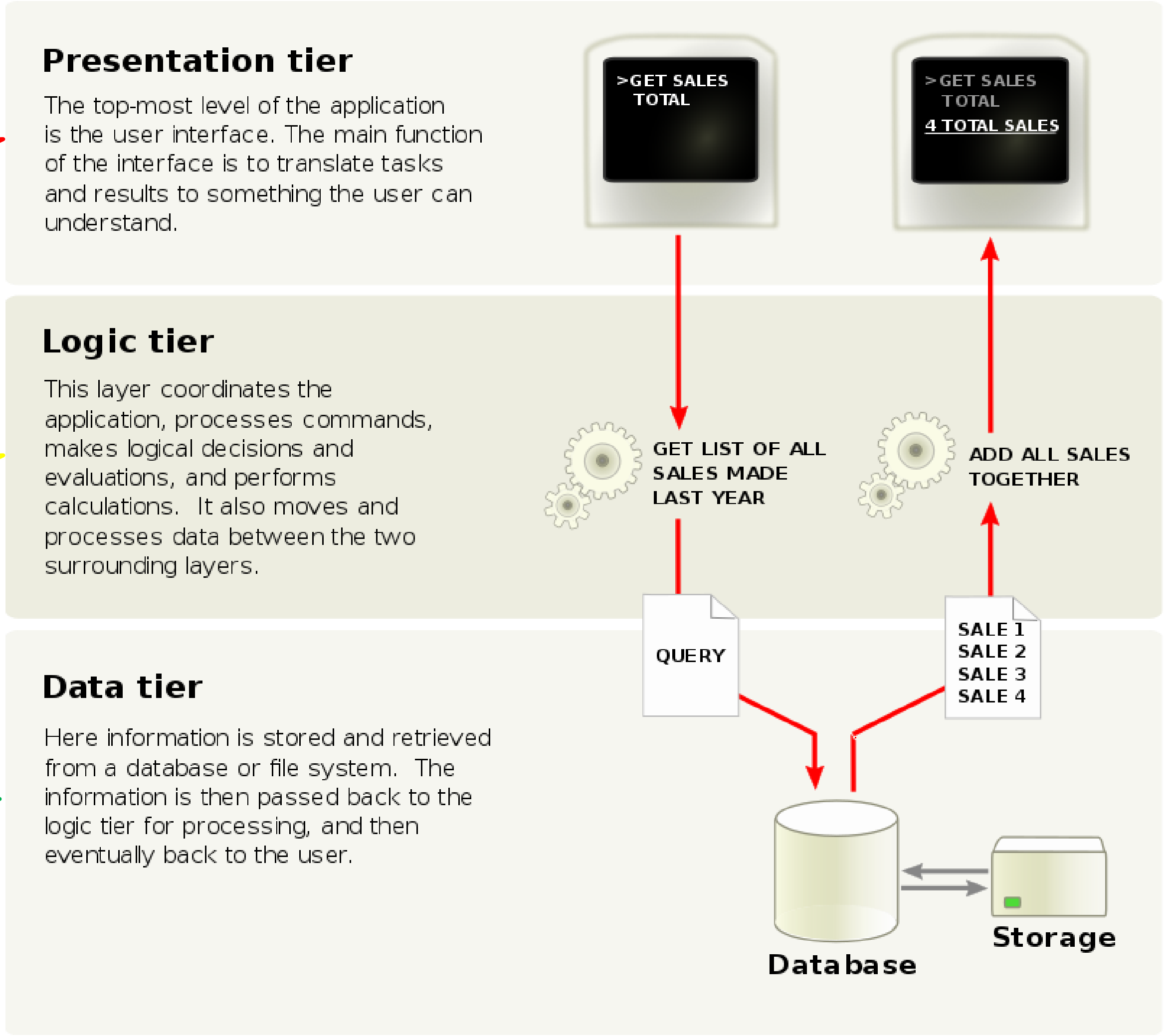
- https://en.wikipedia.org/wiki/Front_and_back_ends

N-tier Architecture

Presentation Layer
Concerned with UI related issues

Business Logic Layer
Data validation, dynamic content processing

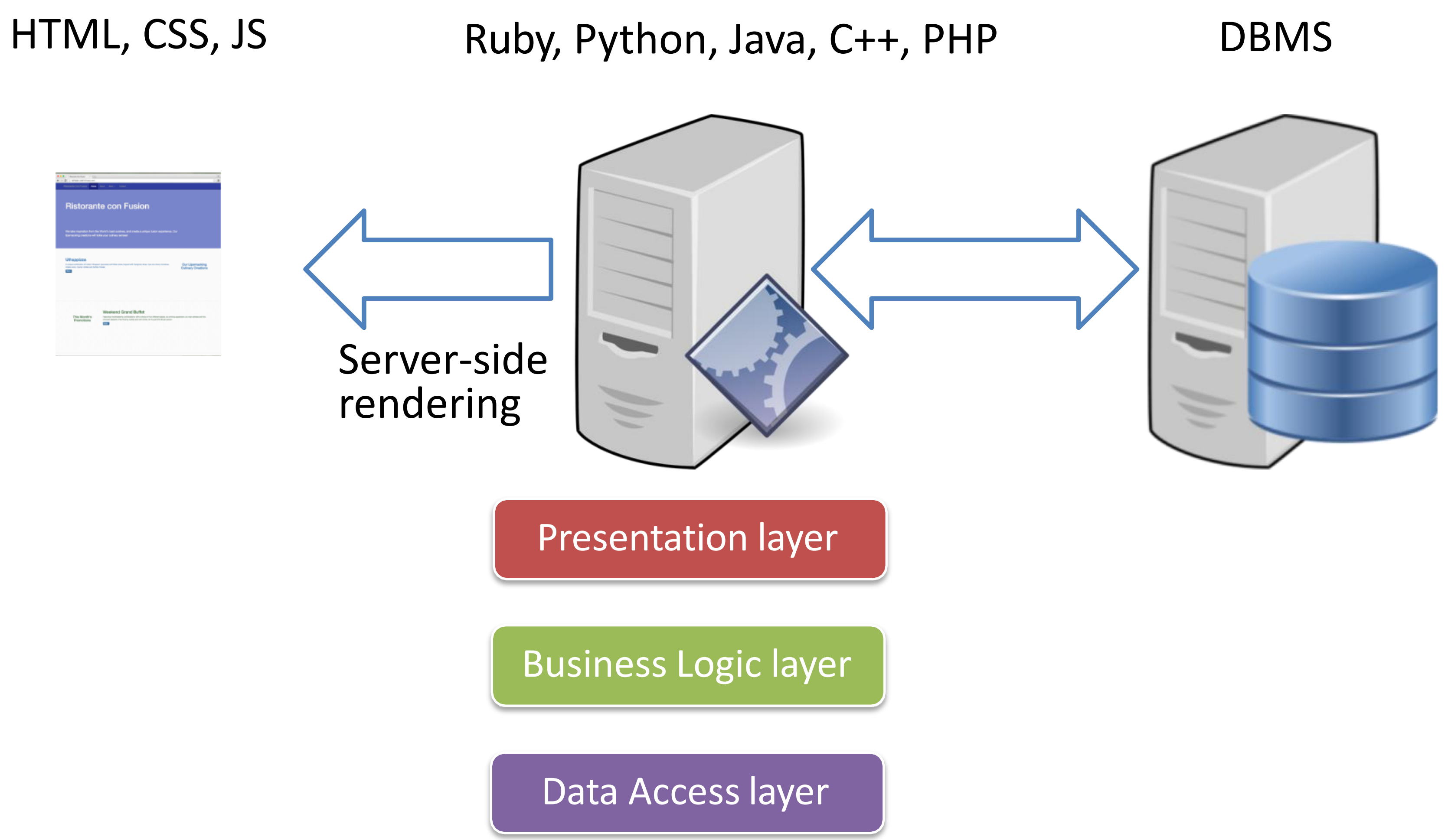
Data Access Layer
Data persistence, data access through an API



Full stack

https://en.wikipedia.org/wiki/Multitier_architecture#Three-tier_architecture

Traditional Web Development

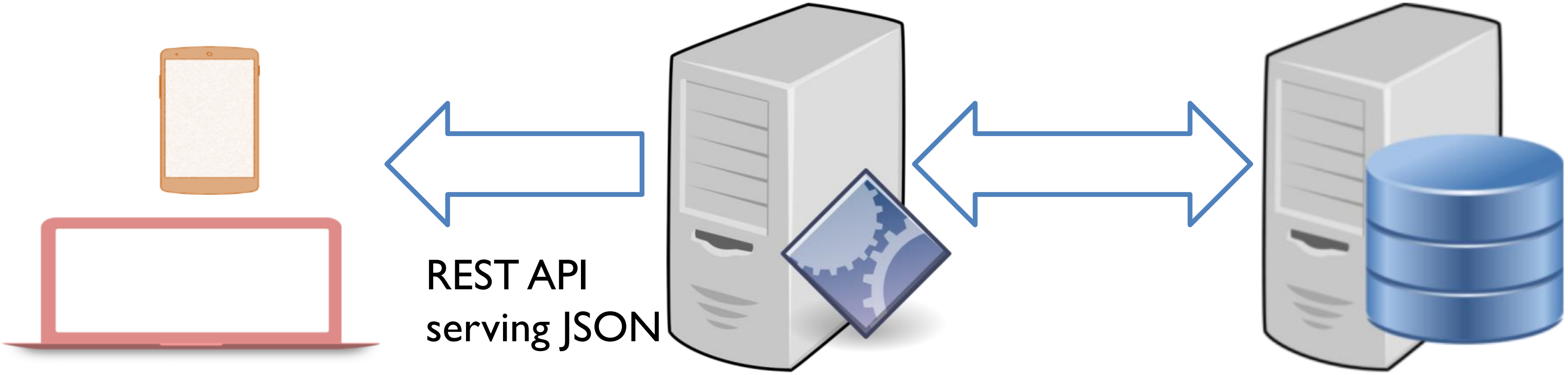


Full Stack JavaScript Development

Single page Apps
using JavaScript frameworks/libraries
like Angular or React

NodeJS and
NodeJS modules

MongoDB
JSON documents

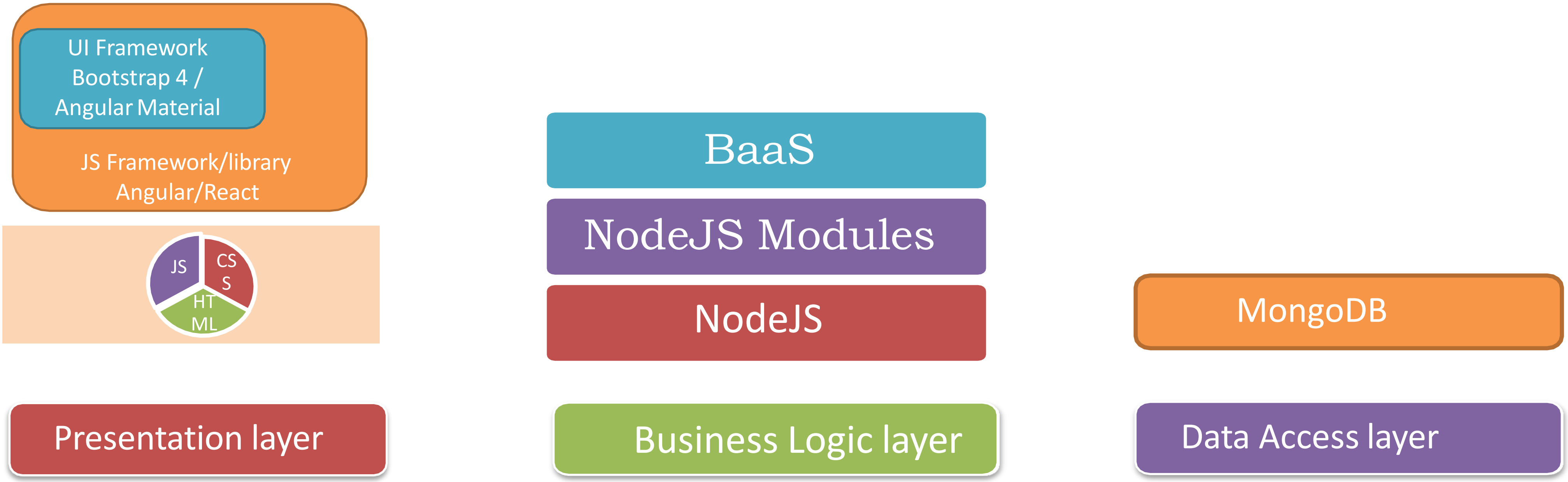


Presentation layer

Business Logic layer

Data Access layer

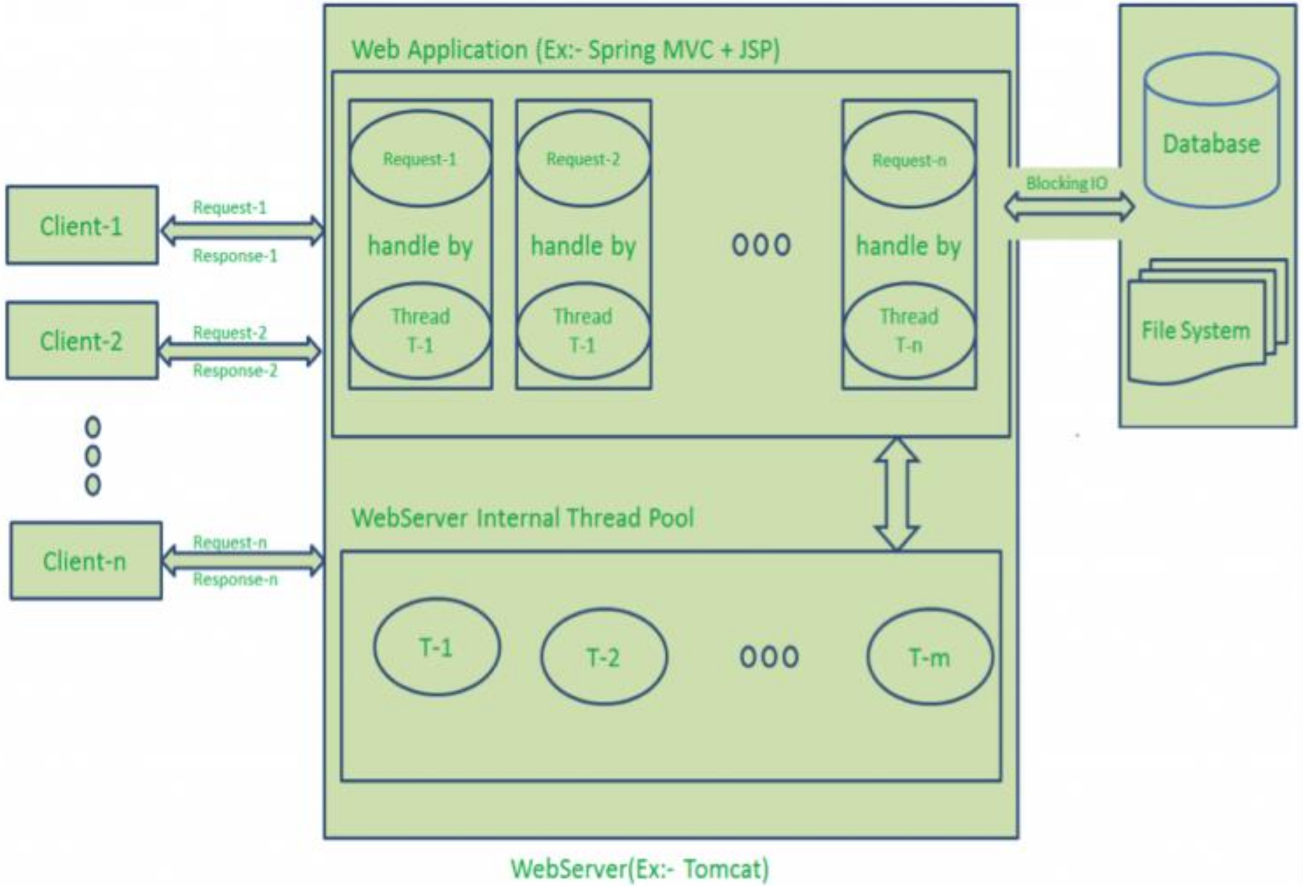
Full Stack Web Development



Node.js Intro

Traditional Web Application Processing Model

- Multi-Threaded Request-Response
  - If "n" is greater than "m" (Most of the times, its true), then server assigns Threads to Client Requests up to available Threads. After all m Threads are utilized, then remaining Client's Request should wait in the Queue until some of the busy Threads finish their Request-Processing Job and free to pick up next Request.
  - If those threads are busy with Blocking IO Tasks (For example, interacting with Database, file system, JMS Queue, external services etc.) for longer time, then remaining clients should wait longer time.



https://www.journaldev.com/7462/node-js-architecture-single-threaded-event-loop

Drawbacks of Request/Response Stateless Model

- Handling more and more concurrent client’s request is bit tough.
- When Concurrent client requests increases, then it should use more and more threads, finally they eat up more memory.
- Sometimes, Client’s Request should wait for available threads to process their requests.
- Wastes time in processing Blocking IO Tasks.

I/O

- **I/O:** A communication between CPU and any other process external to the CPU (memory, disk, network).
- **I/O latency** is defined simply as the time that it takes to complete a single I/O operation.

System Event	Actual Latency	Scaled Latency
One CPU cycle	0.4 ns	**1 s**
Level 1 cache access	0.9 ns	**2 s**
Level 2 cache access	2.8 ns	**7 s**
Level 3 cache access	28 ns	**1 min**
Main memory access (DDR DIMM)	~100 ns	**4 min**
Intel® Optane™ DC persistent memory access	~350 ns	**15 min**
Intel® Optane™ DC SSD I/O	<10 μs	**7 hrs**
NVMe SSD I/O	~25 μs	**17 hrs**
SSD I/O	50–150 μs	**1.5–4 days**
Rotational disk I/O	1–10 ms	**1–9 months**
Internet call: San Francisco to New York City	65 ms[3]	**5 years**
Internet call: San Francisco to Hong Kong	141 ms[3]	**11 years**

https://www.prowesscorp.com/computer-latency-at-a-human-scale/

I/O needs to be done differently

- Consider two scenarios in real word:
- Movie Ticket
  - You are in a queue to get a movie ticket. You cannot get one until everybody in front of you gets one, and the same applies to the people queued behind you.

Synchronously

- Order Food
  - You are in a restaurant with many other people. You order your food. Other people can also order their food, they don't have to wait for your food to be cooked and served to you before they can order. In the kitchen restaurant workers are continuously cooking, serving, and taking orders. People will get their food served as soon as it is cooked.

Asynchronously

Blocking vs non-blocking?

```
const add = (a,b)=>{
  for(let i=0; i<9e27; i++){}
  console.log(a+b);
}

console.log('start');
const A = add(1,2);
const B = add(2,3);
const C = add(3,4);
console.log('end');
```

Blocking methods execute **synchronously**

```
const add = (a,b)=>{
  setTimeout(()=>{
    for(let i=0; i<9e27; i++){}
    console.log(a+b);
  }, 5000);
}
console.log('start');
const A = add(1,2);
const B = add(2,3);
const C = add(3,4);
console.log('end');
```

non-blocking methods execute **asynchronously**

Why JavaScript?

- JavaScript designed specifically to be used with an event loop:
  - Anonymous functions, closures.
  - Only one callback at a time, no need to lock variables.
  - I/O through event callbacks.
- The culture of JavaScript is already geared towards **event-driven programming.**

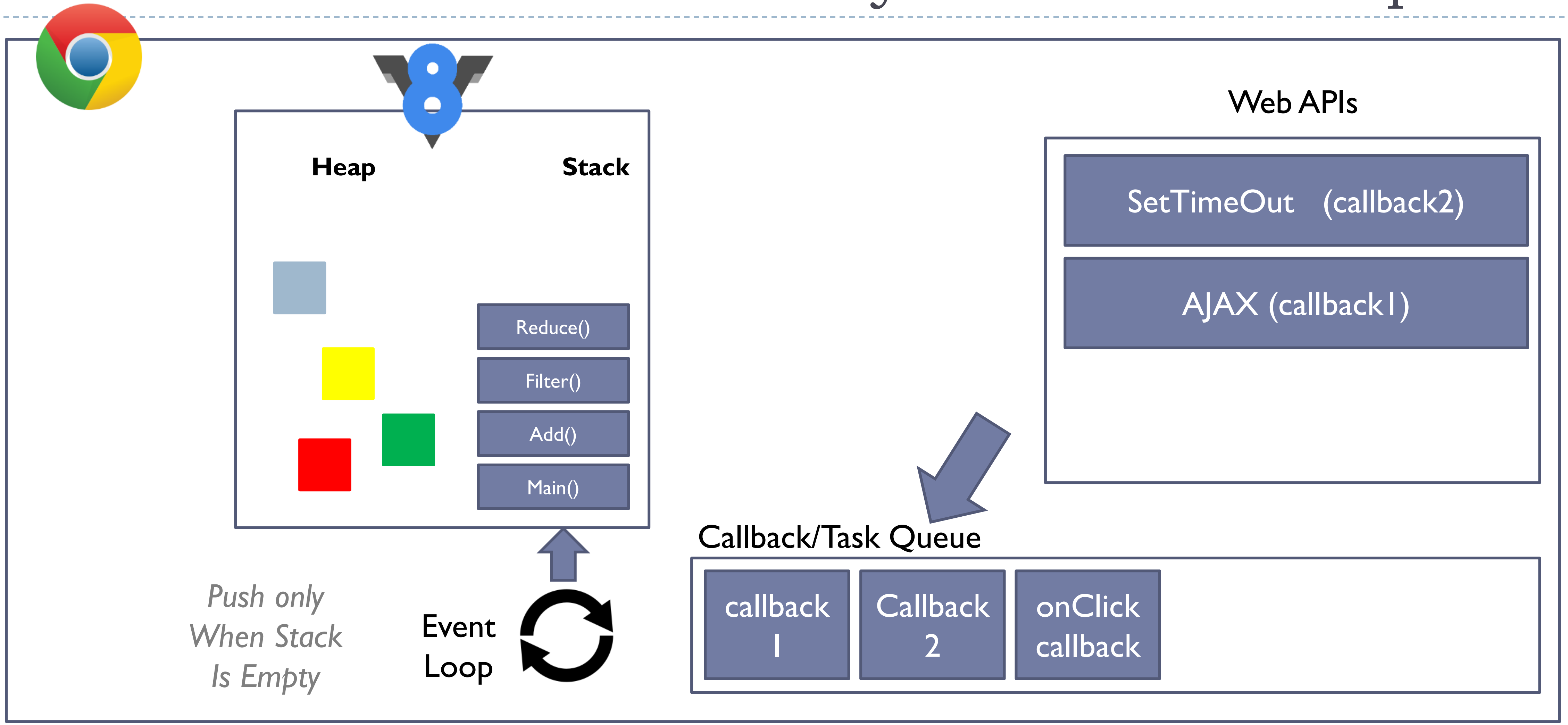
```
puts("Enter your name: ");
var name = gets();
puts("Name: " + name);
```

We're taught to demand input and do nothing until we have it.

```
puts("Enter your name: ");
gets(function (name) {
      puts("Name: " + name);
});
```

Code like this is rejected as too complicated.

Review: Chrome – Concurrency & the Event Loop

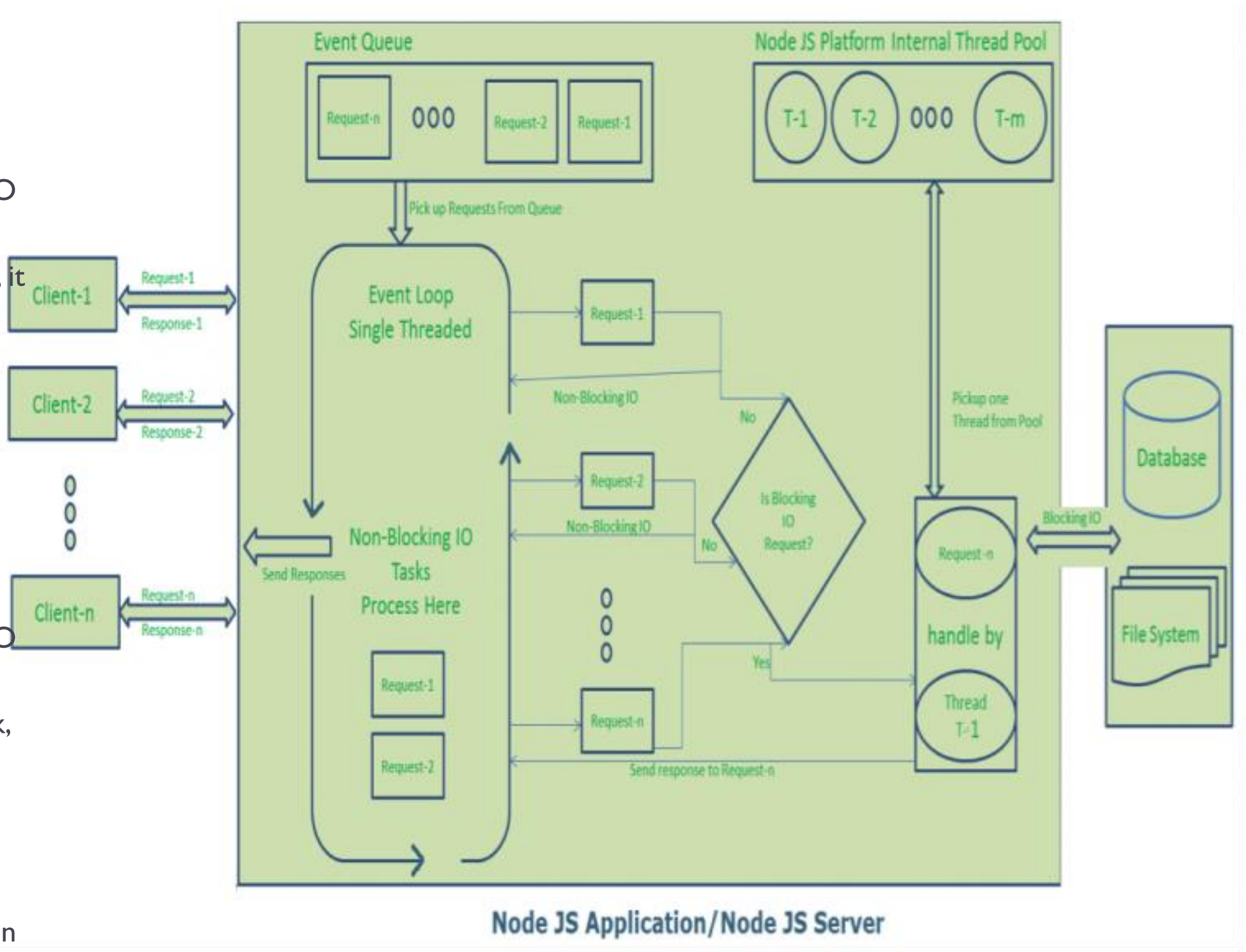


One thing at a time? Not really!

If you block the stack, browser can't do the render queue

Node JS Architecture – Single Threaded Event Loop

- Node JS Processing model mainly based on Javascript Event based model with Javascript callback mechanism.
- Node JS Event Loop Picks up those requests one by one.
- Event 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
- …
- Event 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



Node JS Architecture
- 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

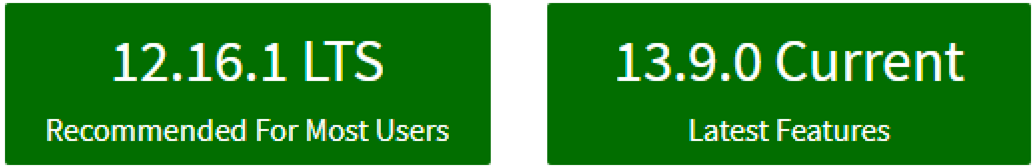
Node.js

- JavaScript runtime built on Chrome V8 JavaScript Engine
- Server-side JavaScript
- Allows script programs do I/O in JavaScript
- Event-driven, non-blocking I/O
- Single Threaded
- CommonJS module system

Setting up Node.js

- Go to nodejs.org and download node. After installing Node we will be able to use it using the command line interface.
  - If Node is installed properly, Try this command: **`node -v`**
  - Hit **`Ctrl+C`** twice or **`Ctrl+D`** once to quit Node.

Node Versions



- **Current**: Under active development. Code for the Current release is in the branch for its major version number (for example, v10.x). Node.js releases a new major version every 6 months, allowing for breaking changes. This happens in April and October every year. Releases appearing each October have a support life of 8 months. Releases appearing each April convert to LTS (see below) each October.

- **LTS**: Releases that receive Long-term Support, with a focus on stability and security. Every even-numbered major version will become an LTS release. LTS releases receive 18 months of *Active LTS* support and a further 12 months of *Maintenance*. LTS release lines have alphabetically-ordered codenames, beginning with v4 Argon. There are no breaking changes or feature additions, except in some special circumstances.

Try these commands

Check number of processors that Node can use

```
node -p "os.cpus()"
```

Check the CPU architecture

```
node -p "process.arch"
```

Check V8 version

```
node -p "process.versions.v8"
```

Check V8 heap

```
node -p "v8.getHeapStatistics()"
```

Check the environment variables

```
node -p "process.env"
```

Node REPL (Read, Eval, Print, Loop)

Run JS scripts

```
node script.js
```

Autocomplete your commands

```
> (tab) (tab)
> global.(tab)
> var a = [];  a.(tab)
```

Underscore: Access to last evaluated value

```
> Math.random(); _
```

The Dot (.) commands

```
.help, .break, .load, .save, .editor
```

First Program

```
setTimeout(function () { console.log("world"); }, 2000); console.log("hello");
```

hello_world.js

```
% node hello_world.js
Hello
// 2 seconds later...
World
```

node.js file name is reserved in Node

Node exits automatically when there is nothing else to do (end of process). Let's change it to never exit, but to keep it in loop!

Node API is not all asynchronous. Some parts of it are synchronous like, for instance, some file operations. Don't worry, they are very well marked: they always end with "Sync". They should only be used when initializing.

The Server Global Environment

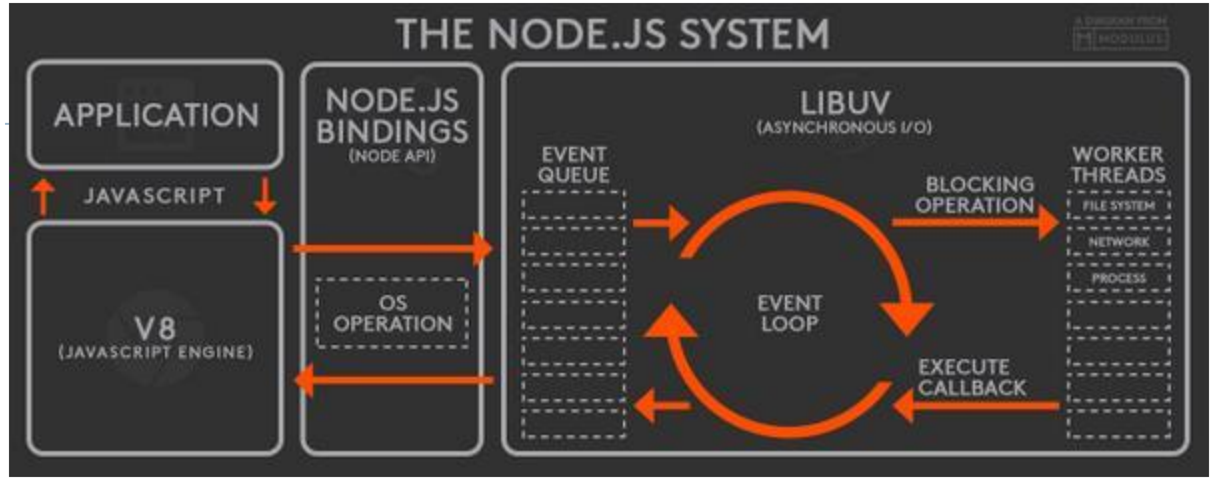
In Node we run JS on the server so we don't have window object. Instead Node provides us with global modules and methods that are automatically created for us *(they aren't part of ECMA specifications)*

- `module`
- `global` *(The global namespace object)*
- `process`
- `require`
- `setInterval`(callback, delay) and `clearInterval`()
- `setTimeout`(callback, delay) and `clearTimeout`()

Global Scope in Node

- Browser JavaScript by default puts everything into its `window` global scope.
- Node.js was designed to behave differently with **everything being local by default**. In case we need to set something globally, there is a `global` object that can be accessed by all modules. (not recommended)
- The document object that represent DOM of the webpage is nonexistent in Node.js.

What's inside Node?

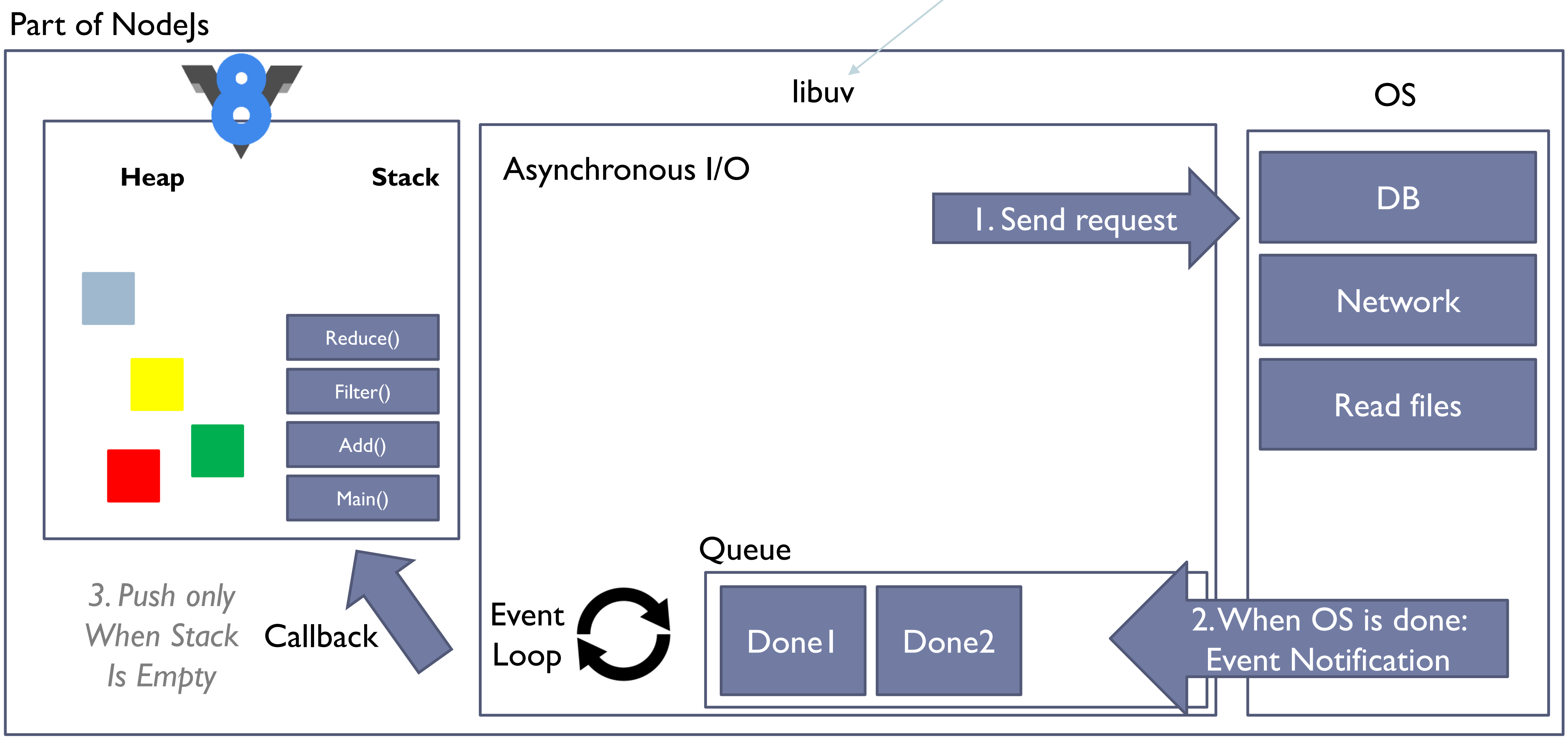


- **V8**
  - Google's open source JavaScript engine.
  - Translates your JS code into machine code
  - V8 is written in C++.
  - Read more about how V8 works here.
- **libuv**
  - a multi-platform support library with a focus on asynchronous I/O.
  - Asynchronous file and file system operations
  - Thread pool
  - …
- **Binding**
  - A wrapper around a library written in one language and expose the library to codes written in another language so that codes written in different languages can communicate.
- **Other Low-Level Components**
  - such as c-ares, http parser, OpenSSL, zlib, and etc, mostly written in C/C++.
- **Application**
  - here is your code, modules, and Node.js' built in modules, written in JS

 https://stackoverflow.com/questions/36766696/which-is-correct-node-js-architecture

JS on the Server

An abstract non-blocking IO operations (Async) using thread pool



What's 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.

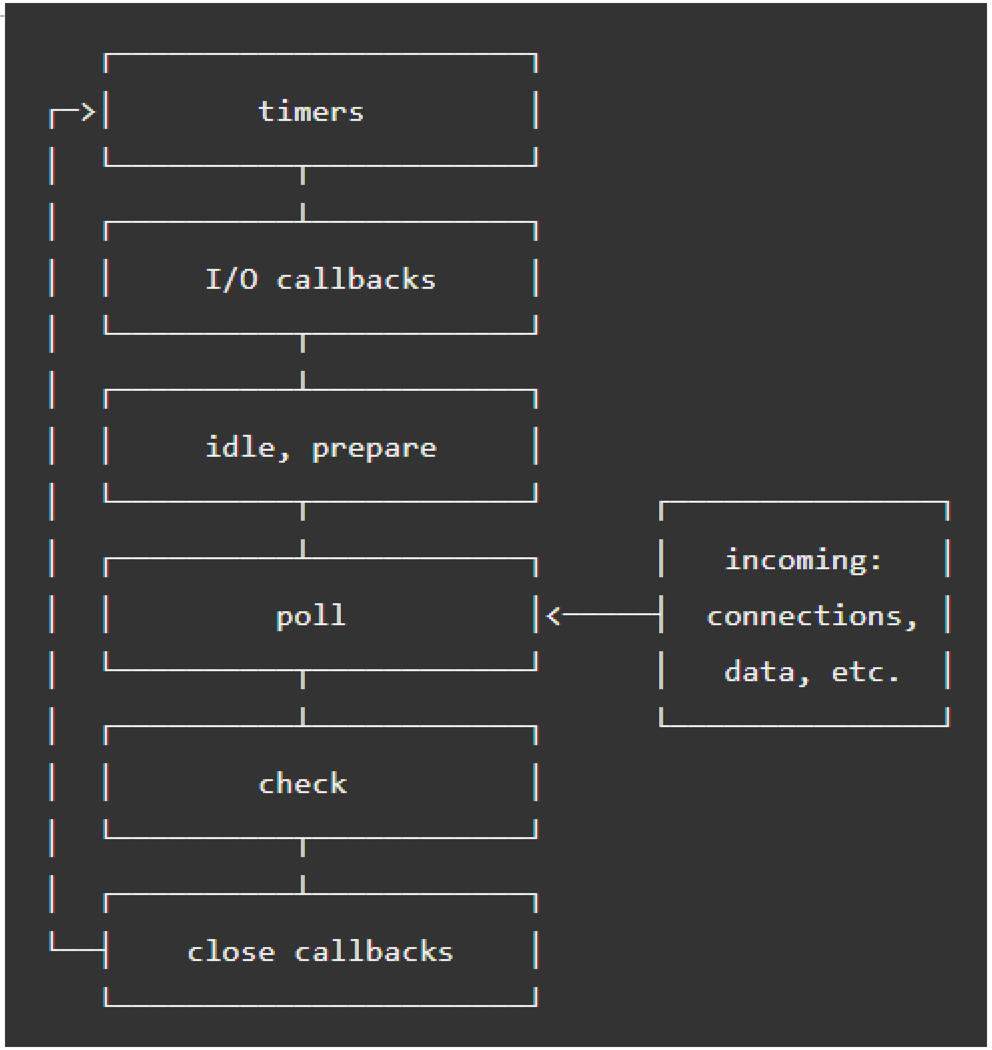
— from node.js doc

A loop that picks events from the event queue and pushes their callbacks into the call stack.

Node.js runs using a **single thread**, at least from a Node.js developer's point of view. Under the hood Node uses many threads through **libuv**.

Event Loop

- **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', ...).



https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/

setTimeout vs setImmediate

- **`setTimeout`**
  - schedules a callback to run after a specific time, the functions are registered in the **timers phase** of the event loop.
- **`setImmediate`**
  - schedules a callback to run at **check phase** of the event loop after IO events' callbacks.

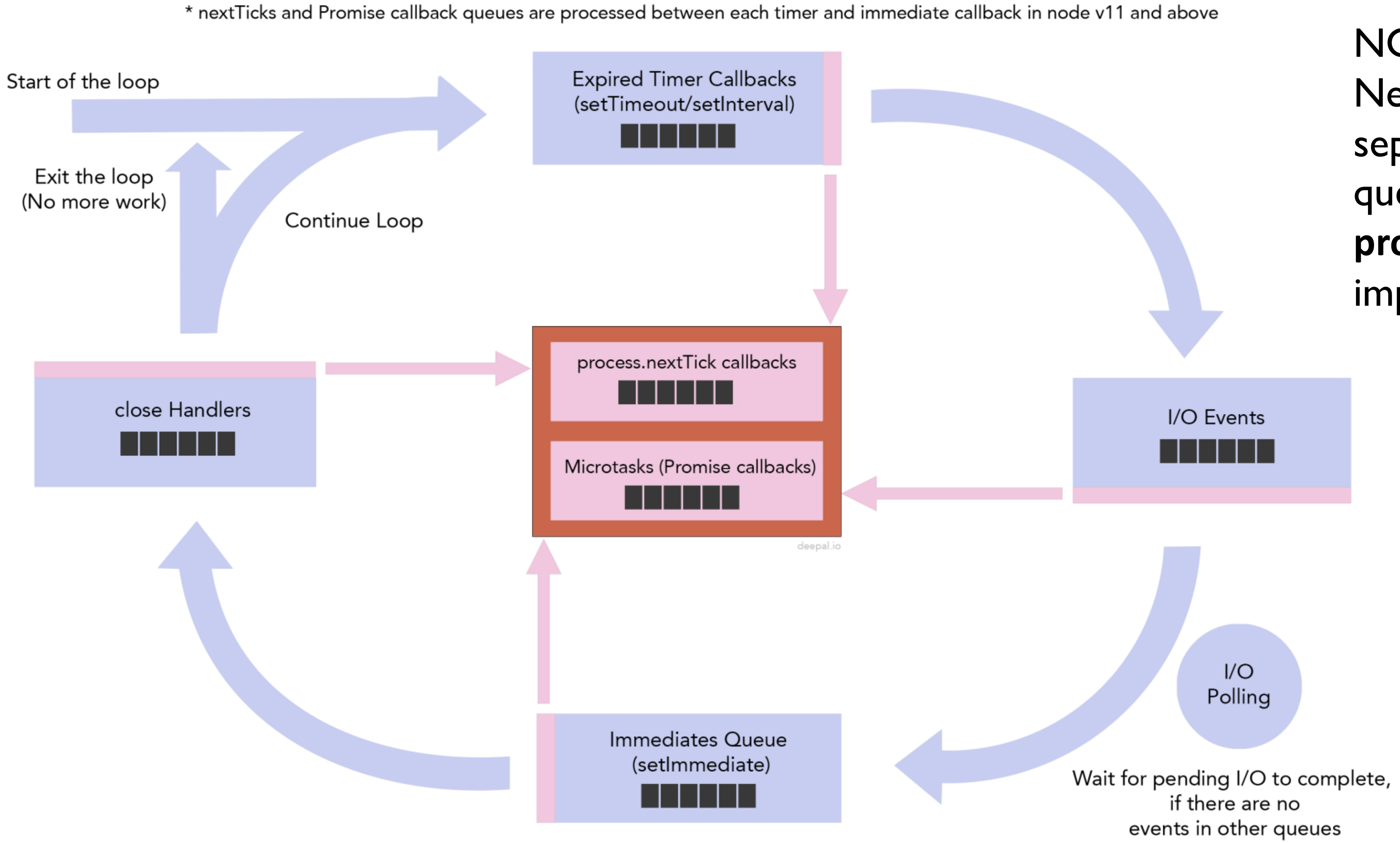
process.nextTick(callback)

`process.nextTick()` is not part of the event loop, it adds the callback into the nextTick queue. Node processes **all the callbacks** in the nextTick queue after the current operation completes and before the event loop continues.

Which means it runs **before** any additional I/O events or timers fire in subsequent ticks of the event loop.

Note: the next-tick-queue is completely drained on each pass of the event loop before additional I/O is processed. As a result, recursively setting `nextTick` callbacks will block any I/O from happening, just like a while(true) loop.

process.nextTick(callback)



NOTE:
Next tick queue is displayed separately from the other four main queues because it is **not natively provided by the libuv**, but implemented in Node.

https://blog.insiderattack.net/event-loop-and-the-big-picture-nodejs-event-loop-part-1-1cb67a182810

process.nextTick(callback)

```
function foo() {
        console.log('foo');
}
process.nextTick(foo);
console.log('bar');
```

Notice that bar will be printed in the console before foo, as we have delayed the invocation of foo() till the next tick of the event loop. We can get the same result by using setTimeout() this way:

```
setTimeout(foo, 0);
console.log('bar');
```

However, process.nextTick() is not just a simple alias to setTimeout(fn, 0).

What's the difference and why it's more efficient?

The Timer Queue in Node.js Event Loop

- Callbacks in the Microtask Queues are executed before callbacks in the Timer Queue.
- Callbacks in microtask queues are executed in between the execution of callbacks in the timer queue.

```
setTimeout(() => console.log('settimeout 1'), 0);
setTimeout(() => {
    console.log('settimeout 2')
    process.nextTick(() => console.log('nextTick inside setTimeout'));
}, 0);
setTimeout(() => console.log('settimeout 3'), 0);

Promise.resolve().then(() => console.log('Promise.resolve 1'));
Promise.resolve().then(() => console.log('Promise.resolve 2'));
process.nextTick(() => console.log('nextTick 1'));
process.nextTick(() => console.log('nextTick 2'));
```

Asynchronous code execution

- Libuv helps handle asynchronous operations in Node.js.
  - For async operations like a network request, libuv relies on the operating system primitives.
  - For async operations like reading a file that has no native OS support, libuv relies on its thread pool to ensure that the main thread is not blocked.

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

console.log('first');
fs.readFile('hello.txt', () => console.log('second'));
console.log('third');
```

I/O Queue in the Node.js Event loop

- Callbacks in the microtask queue are executed before callbacks in the I/O queue.

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

fs.readFile('hello.txt', () => console.log('readFile 1'));
Promise.resolve().then(() => console.log('Promise.resolve 1'));
process.nextTick(() => console.log('nextTick 1'));
```

I/O Polling in the Node.js Event Loop

- The readFile() callback is not queued up at the same as other callbacks. The event loop has to poll to check if I/O operations are complete, and it only queued up completed operation callbacks. This means that when the control enters the I/O queue for the first time, the queue is still empty.
- I/O events are polled and callback functions are added to the I/O queue only after the I/O is complete.

```
const fs = require('fs');
fs.readFile('hello.txt', () => console.log('readFile'));
setTimeout(() => console.log("this is setTimeout"), 0);
setImmediate(() => console.log("this is setImmediate"), 0);
Promise.resolve().then(() => console.log('Promise.resolve 1'));
process.nextTick(() => console.log('nextTick 1'));

for (let i = 0; i < 200000000; i++) { }
```

Check Queue in the Node.js Event loop

- The anomaly is due to how a minimum delay is set for timers.
- If we pass in 0 milliseconds, the interval is set to max(1,0) which is 1. This will result in setTimeout with a 1 milliseconds delay.

- When running setTimeout with a delay 0ms and an I/O async method, the order of execution can never be guaranteed.

```
setTimeout(() => console.log("this is setTimeout..."), 0);
setImmediate(() => { console.log('immediate'); });
```

setTimeout vs setImmediate vs process.nextTick

- `setTimeout(() => { console.log('timeout'); }, 0);`
- `setImmediate(() => { console.log('immediate'); });`
- `process.nextTick(()=> console.log('nexttick'));`

- What's the output of this code and why?

Close Queue in the Node.js Event Loop

- Close queue callbacks are executed after all other queue callbacks in a given iteration of the event loop.

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

const rd = fs.createReadStream("input.txt");
rd.close();
rd.on("close", () => console.log('readablStream close event'))
setTimeout(() => console.log("this is setTimeout"), 0);
setImmediate(() => console.log("this is setImmediate"), 0);
Promise.resolve().then(() => console.log('Promise.resolve 1'));
process.nextTick(() => console.log('nextTick 1'));
```