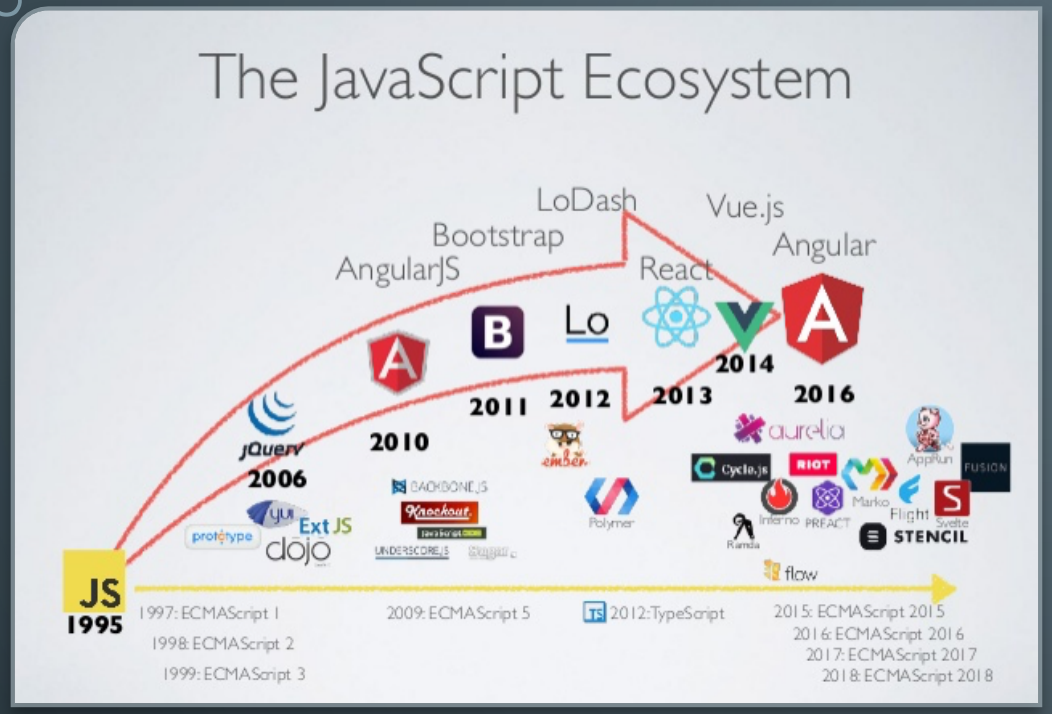
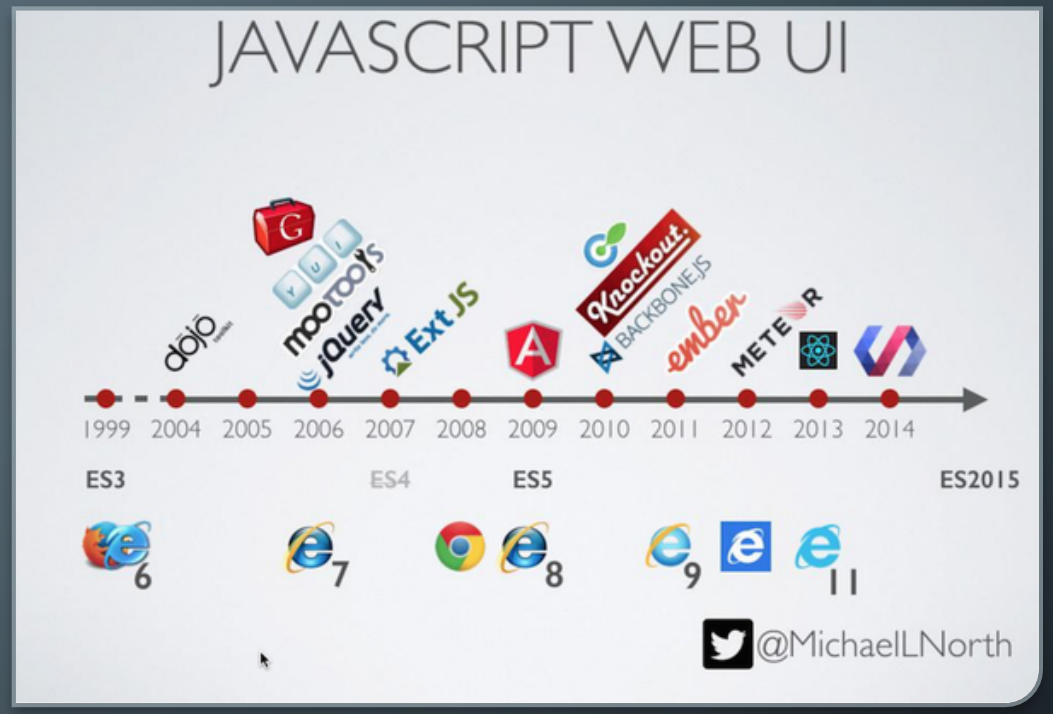
NODE.JS, BUT WHY ?

1. WHAT IS UI AND WHAT IS NODE ?
2. WHY NODE ? WHERE IT CAN BE USED ?
3. HOW IT WORKS ?
4. DEMO FOR EVENT LOOP .
5. DIFFERENCE BETWEEN NODE AND OTHER BACK END TECHNOLOGY LIKE JAVA.





NODE.JS

NODE.JS® IS A JAVASCRIPT RUNTIME BUILT ON CHROME'S V8 JAVASCRIPT ENGINE.

WHAT IS NODE.JS ?

Node.js was written initially by *Ryan Dahl* in 2009. Written in C (libuv), C++(V8 & Addon) , JavaScript.

Node.js is an *open-source, cross-platform,* JavaScript runtime environment that executes JavaScript code outside of a web browser.

Node.js operates on a single-thread event loop, using non-blocking I/O calls, allowing it to support tens of thousands of concurrent connections without incurring the cost of thread context switching.

The design of sharing a single thread among all the requests that use the observer pattern is intended for building highly concurrent applications, where any function performing I/O must use a callback.

To accommodate the single-threaded event loop, Node.js uses the libuv library—which, in turn, uses a fixed-sized thread pool that handles some of the non-blocking asynchronous I/O operations.

In January 2010, a package manager was introduced for the Node.js environment called *npm*.

WHY NODE.JS ?

Dahl criticized the limited possibilities of the most popular web server in 2009, Apache HTTP Server, to handle a lot of concurrent connections (up to 10,000 and more) and the most common way of creating code (sequential programming), when code either blocked the entire process or implied multiple execution stacks in the case of simultaneous connections.

Problems that Node aims to solve:

- How to serve many thousands of simultaneous clients *efficiently*.
- Scaling networked applications beyond a single server.
- Preventing I/O operations from becoming bottlenecks.
- Eliminating single points of failure, thereby ensuring reliability.
- Achieving parallelism safely and predictably.

DESIGN PRINCIPLE FOR NODE.JS

The general principle is operations(I/O) must never block. Node's desire for speed (high concurrency) and efficiency (minimal resource usage) demands the reduction of waste. A waiting process is a wasteful process, especially when waiting for I/O.

Dahl was guided by a few rigid principles:

• A Node program/process runs on a single thread, ordering execution

through an event loop

• Web applications are I/O intensive, so the focus should be on making I/O fast

• Program flow is always directed through asynchronous callbacks

• Expensive CPU operations should be split off into separate parallel processes,

emitting events as results arrive

• Complex programs should be assembled from simpler programs

WHERE TO USE NODE.JS ?

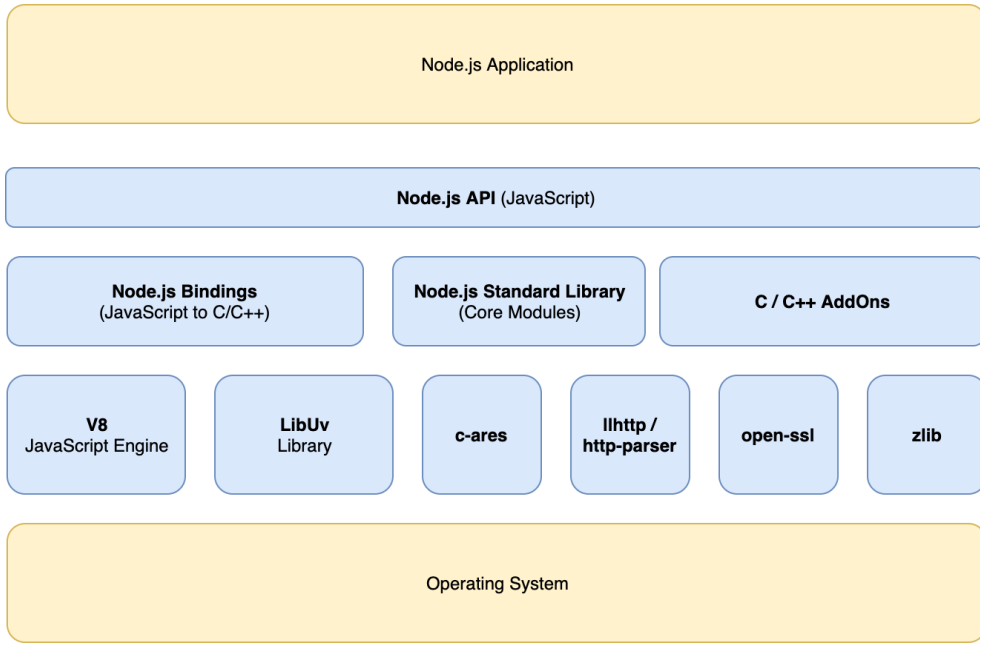
Node.js brings event-driven programming to web servers, enabling development of fast web servers in JavaScript.

Developers can create scalable servers without using threading, by using a simplified model of event-driven programming that uses callbacks to signal the completion of a task.

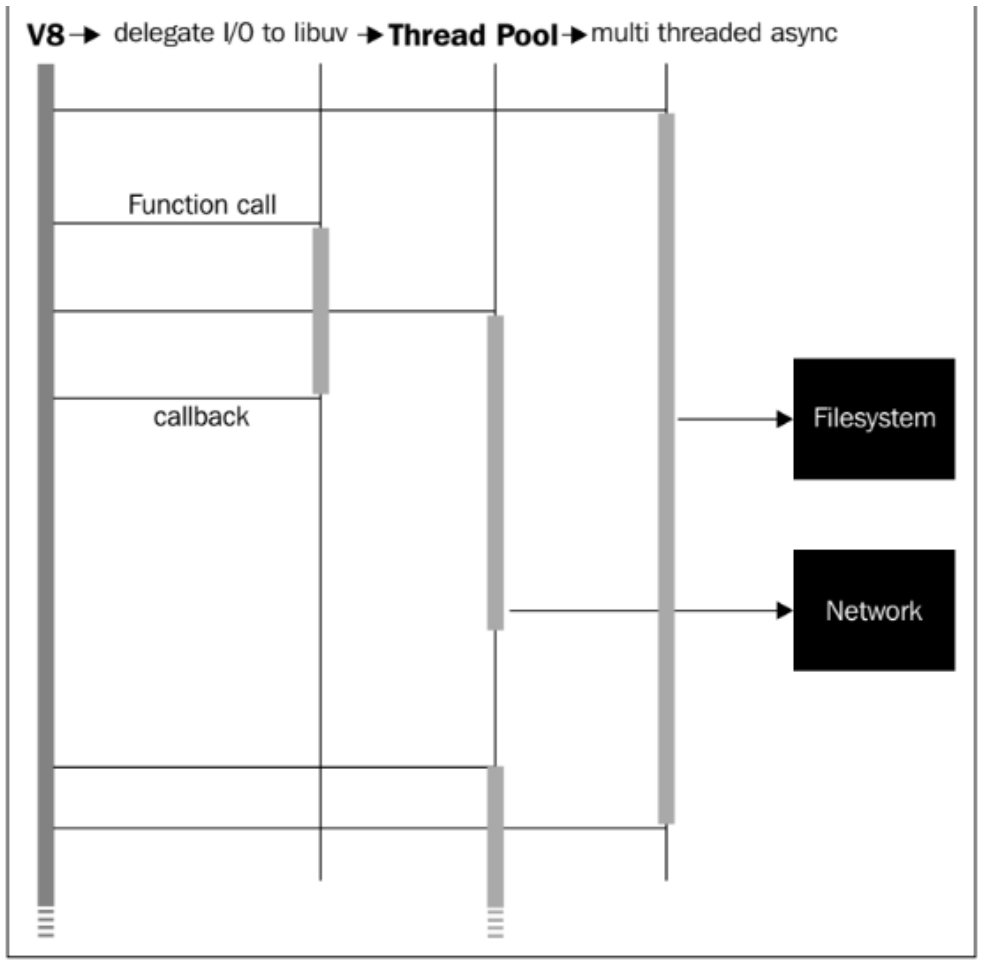
Node.js connects the ease of a scripting language (JavaScript) with the power of Unix network programming.

NodeJS simply provides **non-blocking asynchronous I/O model** even with **single thread**. This makes NodeJS more suitable for I/O intensive applications than other available options. (Note CPU intensive applications are not suited for NodeJS being its single threaded, and will block execution.)

NODE.JS ARCHITECTURE

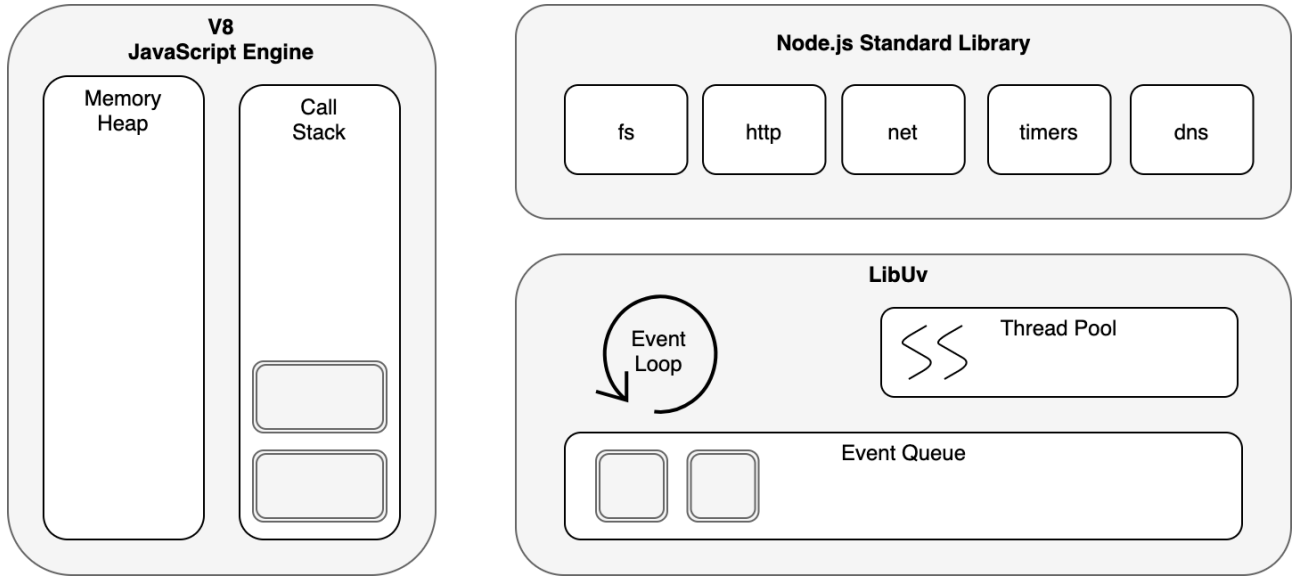


- **V8 JavaScript Engine**: Consists of Memory Heap, Call Stack, Garbage Collector and converts JavaScript code into machine code of given OS.
- **LibUv**: Consists of Thread Pool and handles Event Loop, Event Queue. Its multi-platform C library focusing on asynchronous I/O operations.
- **Node.js API**: Exposed JavaScript API to be used by applications
- **Node.js Standard Library**: Consists of libraries operating system related functions for Timers setTimeout, File System fs, Network Calls http.
- **llhttp**: parsing HTTP request/response (Previously http-parse used)
- **C - ares**: C library for async DNS request used in dns module.
- **open-ssl**: Cryptographic functions used in tls (ssl), crypto modules.
- **zlib**: Interface to compress and decompress by sync, async and streaming.



HOW IT WORKS ?

- Node itself will efficiently manage I/O operations; its process object refers to the V8 runtime.
- When I/O operations are initiated within this loop they are delegated to libuv.
- Libuv manages the request using its own (multi-threaded, asynchronous) environment.
- libuv announces the completion of I/O operations, allowing any callbacks waiting on this event to be re-introduced to the main V8 thread for execution.



EVENT LOOP & CALL STACK

- The event loop is what allows Node.js to perform non-blocking I/O operations even though JavaScript is single-threaded — by offloading operations to the system kernel whenever possible
- Event Loop comes into play here, where it'll move callback functions from Event Queue to Call Stack to be executed by main thread. (Event Loop and Call Stack is run by main thread.)
- When Call Stack is empty and Event Queue have pending functions, Event Loop moves event and its callback from Event Queue to Call Stack and will be executed by main thread.
- For callback function to be executed by main thread, it should be moved to Call Stack
- Call stack is responsible for executing the current function called by main thread or pushed from event que.
- This Event Queue consists of all callback functions of completed functions that are waiting to be executed by main thread.

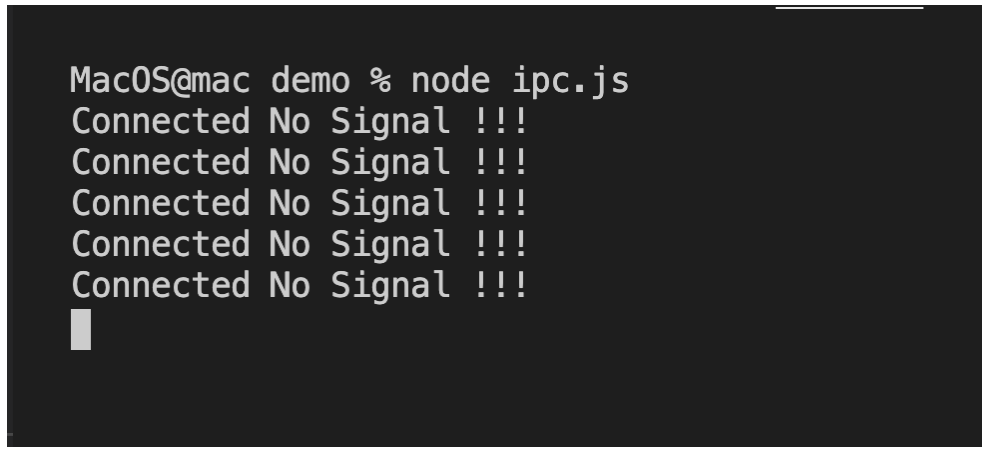
EVENT LOOP WORKING AS V8 & LIBUV PROCESS

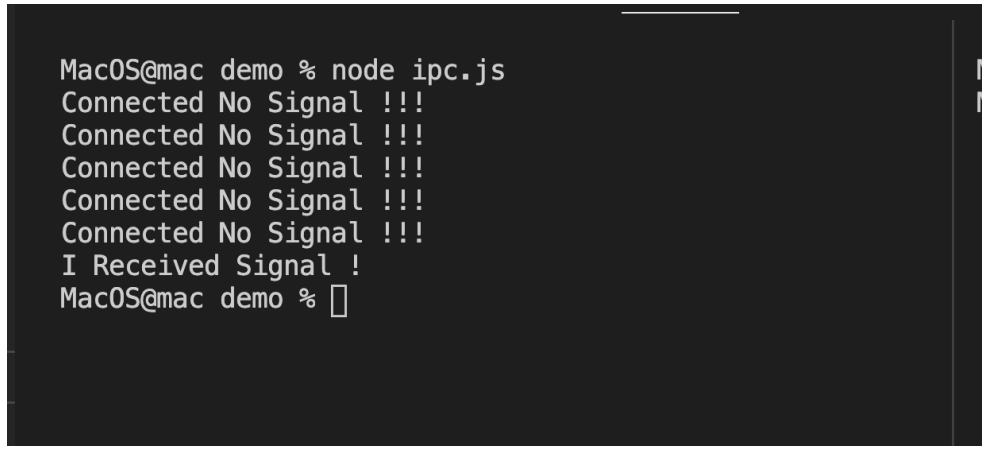
ipc.js

```
// Simple JavaScript setInterval Execution in Nodejs
const interval =  setInterval(
    () =>  process.stdout.write("Connected No Signal !!!\n") , 2000);

/************************ Abstaraction to Nodejs Programmer ********************/

// Call back is just a function executed in callstack from eventque.
const callback = () => {
    process.stdout.write(`I Received Signal ! \n`);
    // removing the setIntrval registration
    // process exits as no more registration
    // & event callback in queue.
    clearInterval(interval);
}


// Observer Pattern Implememted on Process Object
// It is main thread refers to V8
// Can handle POSIX signals as event in Nodejs
process.on('SIGUSR1', callback);

```

signal.sh

```
#!/bin/zsh
PIDS=$(ps -ef | grep ipc.js | awk '{print $2}');
read -r PID <<<${PIDS};
kill -s SIGUSR1 ${PID}

```





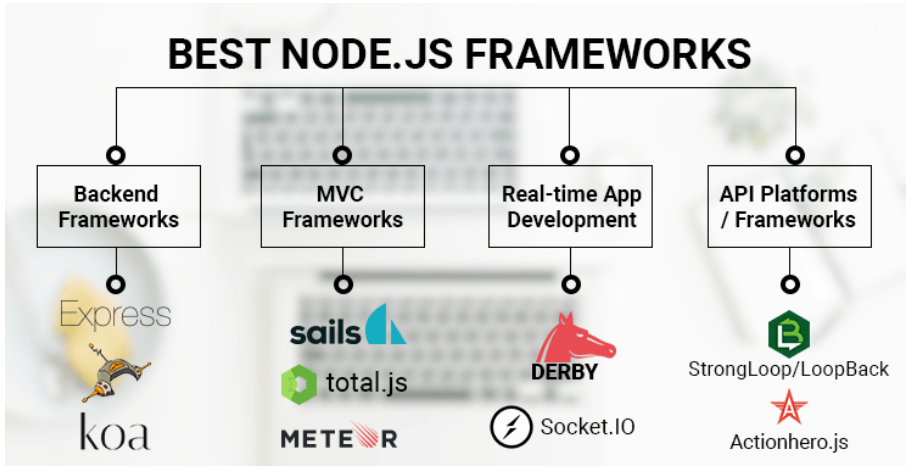
THE EVENT LOOP IS COMMUNICATION B/W V8 & LIBUV

- Running *ipc.js* from terminal it starts a node process(main thread) which encounters a setInterval registration it and inform libuv to start event loop.
- When we try to send a signal via *signal.sh* to this process and callback clears the setInterval registration there are no more registration and callback in queue, event loop and hence main thread(V8 runtime) quits ending node process.
- That's how we can say how they communicate behind the seen. Event loop is not just the events in queue rather its a continuous observation between V8 & libuv which orchestrate all delegation and callback execution.

Network and I/O	Strings and Buffers	Utilities
TTY	Path	Utilities
UDP/Datagram	Buffer	VM
HTTP	Url	Readline
HTTPS	StringDecoder	Domain
Net	QueryString	Console
DNS		Assert
TLS/SSL		
Readline		
FileSystem		
Encryption and Compression	**Environment**	**Events and Streams**
ZLIB	Process	Child Processes
Crypto	OS	Cluster
PunyCode	Modules	Events
		Stream

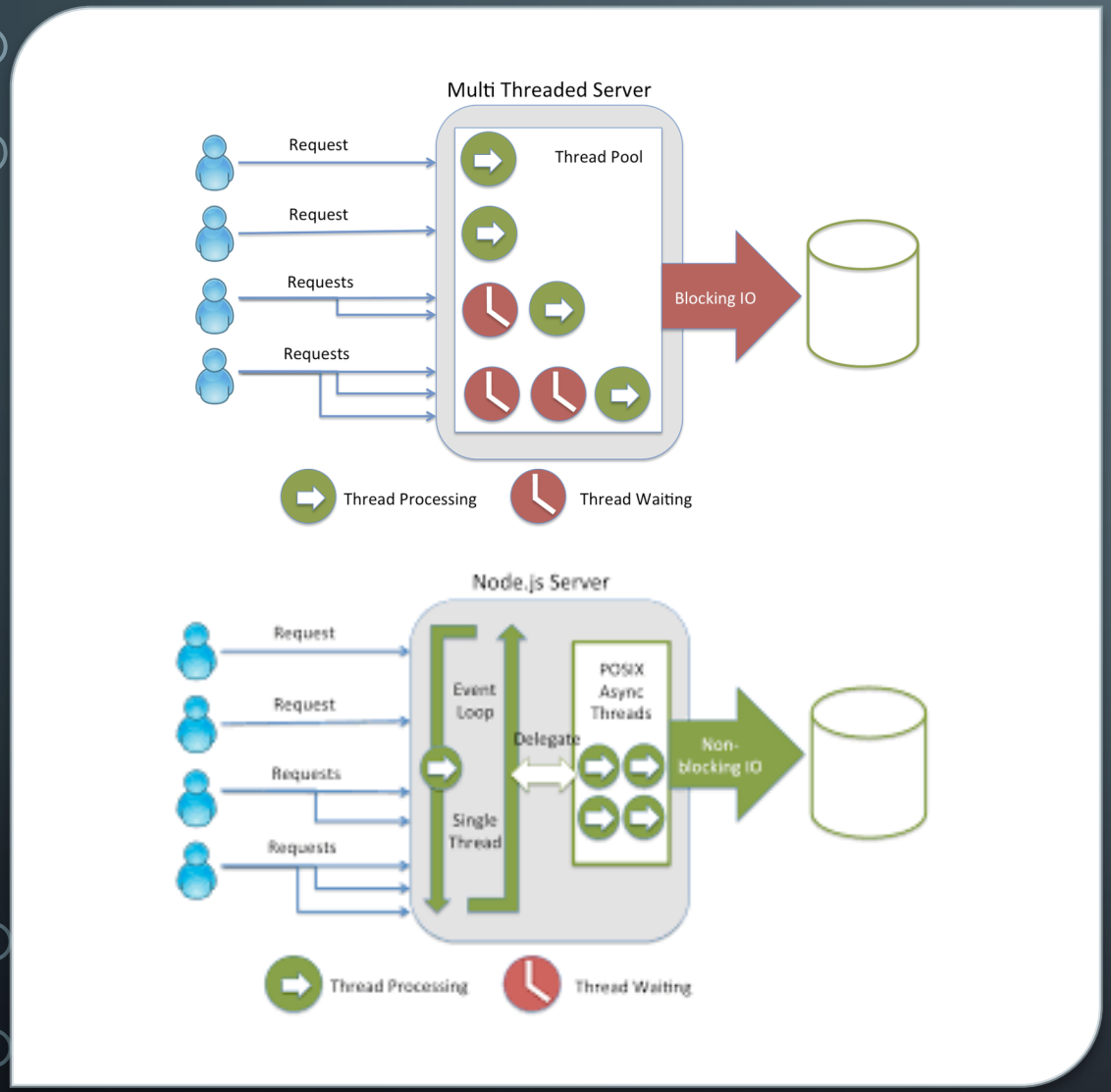
CORE MODULES

- Process
- Http/Https
- Filesystem
- Events
- Stream
- Buffer
- Path
- Cluster



JAVASCRIPT FRAMEWORKS FOR NODE.JS

- Express is API backend for http and act as utility to handle Http with ease.
- Nest is Typescript based framework fast, reliable and wraps the Express to provide Spring like syntax.
- Socket.IO is a wrapper to node WebSocket module to build socket-based applications.



JAVA VS NODE.JS

- Where Java wins: Rock-solid foundation
- Where Node.js wins: Ubiquity
- Where Java wins: Better IDEs
- Where Node.js wins: Database queries
- Where Java wins: Types
- Where Node.js wins: Syntactic flexibility
- Where Java wins: Simple build process
- Where Node.js wins: Desktop
- Where Java wins: Handhelds
- Where Node.js wins: JSON
- Where Java wins: Remote debugging