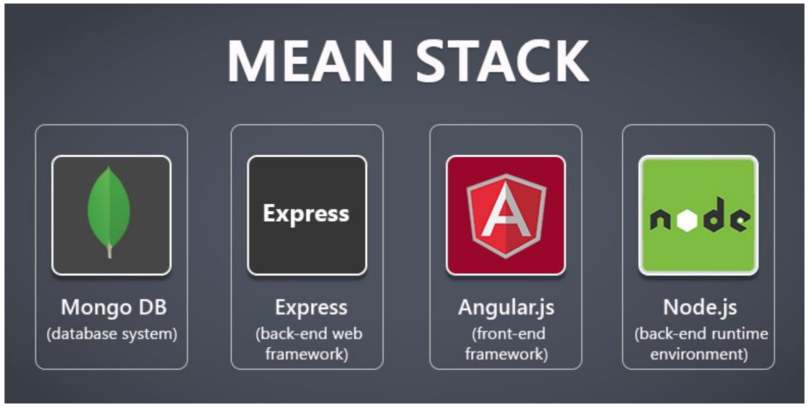
NodeJS

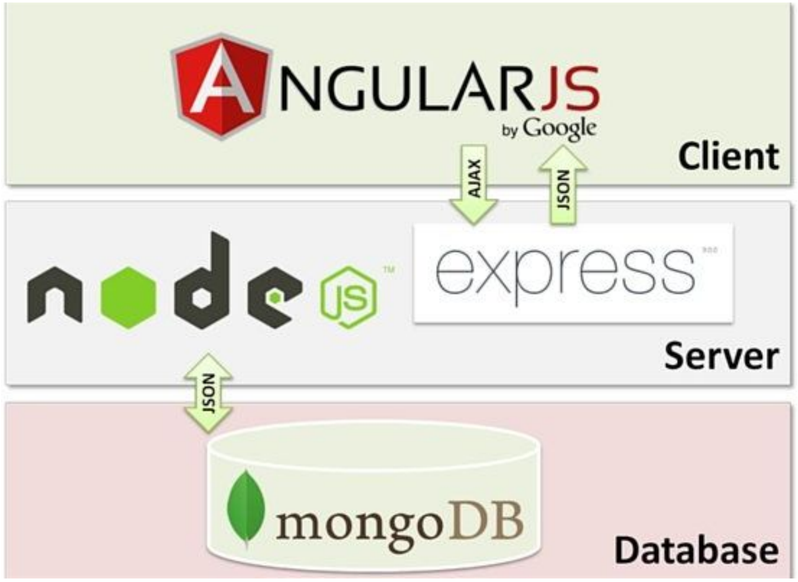
JavaScript Everywhere

MEAN

- M = mongoDB
- E = Express
- A = Angular.js
- N=Node.js

FULL stack solution





What is node.js ?

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

- Node.js was written and introduced by Ryan Dahl in 2009.

Node.js official web site: https://nodejs.org

Node.json github: https://github.com/nodejs/node

Node.jscommunityconference http://nodeconf.com

Consistancy

- Use of JS on both the client and server-side should remove need to "context switch"
  - Client-side JS makes heavy use of the DOM, no access to files/databases
  - Server-side JS deals mostly in files/databases, no DOM
    - JSDom project for Node works for simple tasks, but not much else

System Requirements

Node runs best on the POSIX-like operating systems. These are the various UNIX derivatives (Solaris, and so on) or work a likes (Linux, Mac OS X, and so on).

While Windows is not POSIX compatible, Node can be built on it either using POSIX compatibility environments (in Node 0.4x and earlier).

Advantages node.js ?

- Node.js is open-source.
- Uses JavaScript to build entire server side application.
- Lightweight framework that includes bare minimum modules. Other modules can be included as per the need of an application.
- Asynchronous by default. So it performs faster than other frameworks.
- Cross-platform framework that runs on Windows, MAC or Linux

Why Use Node.js ?

- Node's goal is to provide an easy way to build scalable **network programs**.
- Node provides JavaScript API to access the network and file system.
- Nodejs is not server side scripting that uses server (PHP uses Apache). It is used to create server that can handle request.
- Node.js can be used to build different types of applications such as
  - command line application
  - web application
  - real-time chat application
  - High Concurreny Applications
  - Game servers (Fast and high-performance servers that need to processes thousands of requests at a time)
  - Streaming server (clients have request's to pull different multimedia contents from this server.)
  - **REST API server** etc.

When to use it ?

- Chat/Messaging
- Real-time Applications
- Intelligent Proxies
- High Concurrency Applications
- Communication Hubs
- Coordinators

Node.js for….

- Web application
- Websocket server
- Ad server
- Proxy server
- Streaming server
- Fast file upload client
- Any Real-time data apps
- Anything with high I/O

Node.js VS Apache

1. It's faster
2. It can handle tons of concurrent requests

Platform	Number of request per second
PHP (via Apache)	3187,27
Static (via Apache)	2966,51
Node.js	5569,30

Who Uses Node.js ?

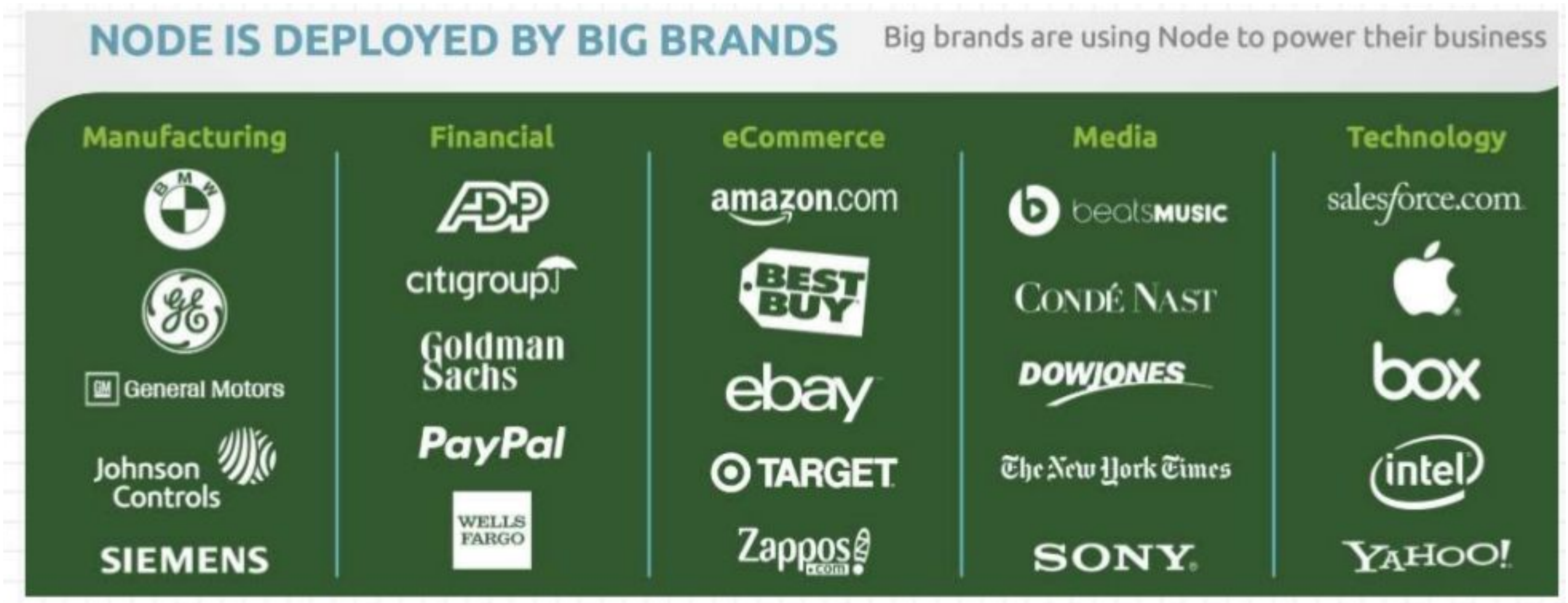
Paypal – A lot of sites within Paypal have also started the transition onto Node.js.

LinkedIn - LinkedIn is using Node.js to power their Mobile Servers, which powers the iPhone, Android, and Mobile Web products.

Mozilla has implemented Node.js to support browser APIs which has half a billion installs.

Ebay hosts their HTTP API service in Node.js

Who Uses Node.js ?


NODE IS DEPLOYED BY BIG BRANDS

Big brands are using Node to power their business

Manufacturing	Financial	eCommerce	Media	Technology
BMW	ADP	amazon.com	beatsMUSIC	salesforce.com
GE	citigroup	BEST BUY	CONDÉ NAST	
General Motors	Goldman Sachs	ebay	DOWJONES	box
Johnson Controls	PayPal	TARGET	The New York Times	intel
SIEMENS	WELLS FARGO	Zappos.com	SONY	YAHOO!

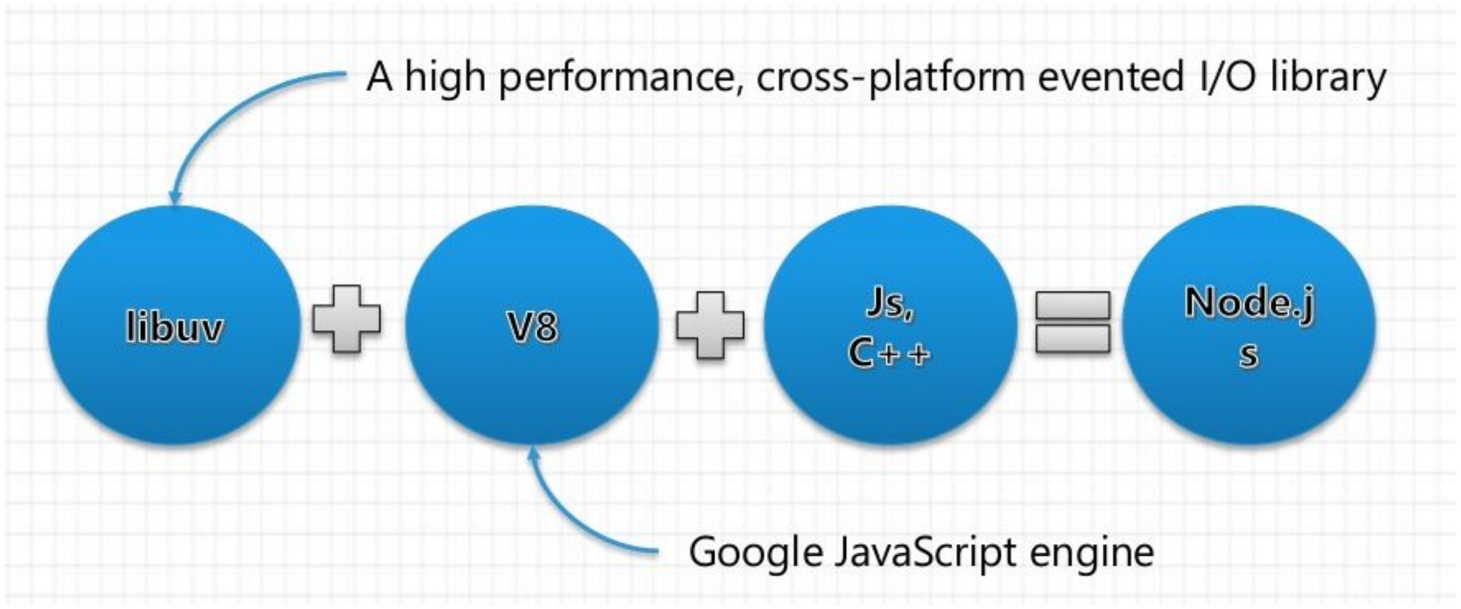
What can’t do with Node?

- Node is a platform for writing JavaScript applications outside web browsers. This is not the JavaScript we are familiar with in web browsers. There is no DOM built into Node, nor any other browser capability.
- Node can’t run on GUI, but run on terminal
- CPU intensive work : carry out some long-running calculations in the background, it won't be able to process any other requests. As discussed above, Node.js is used best where processing needs less CPU time

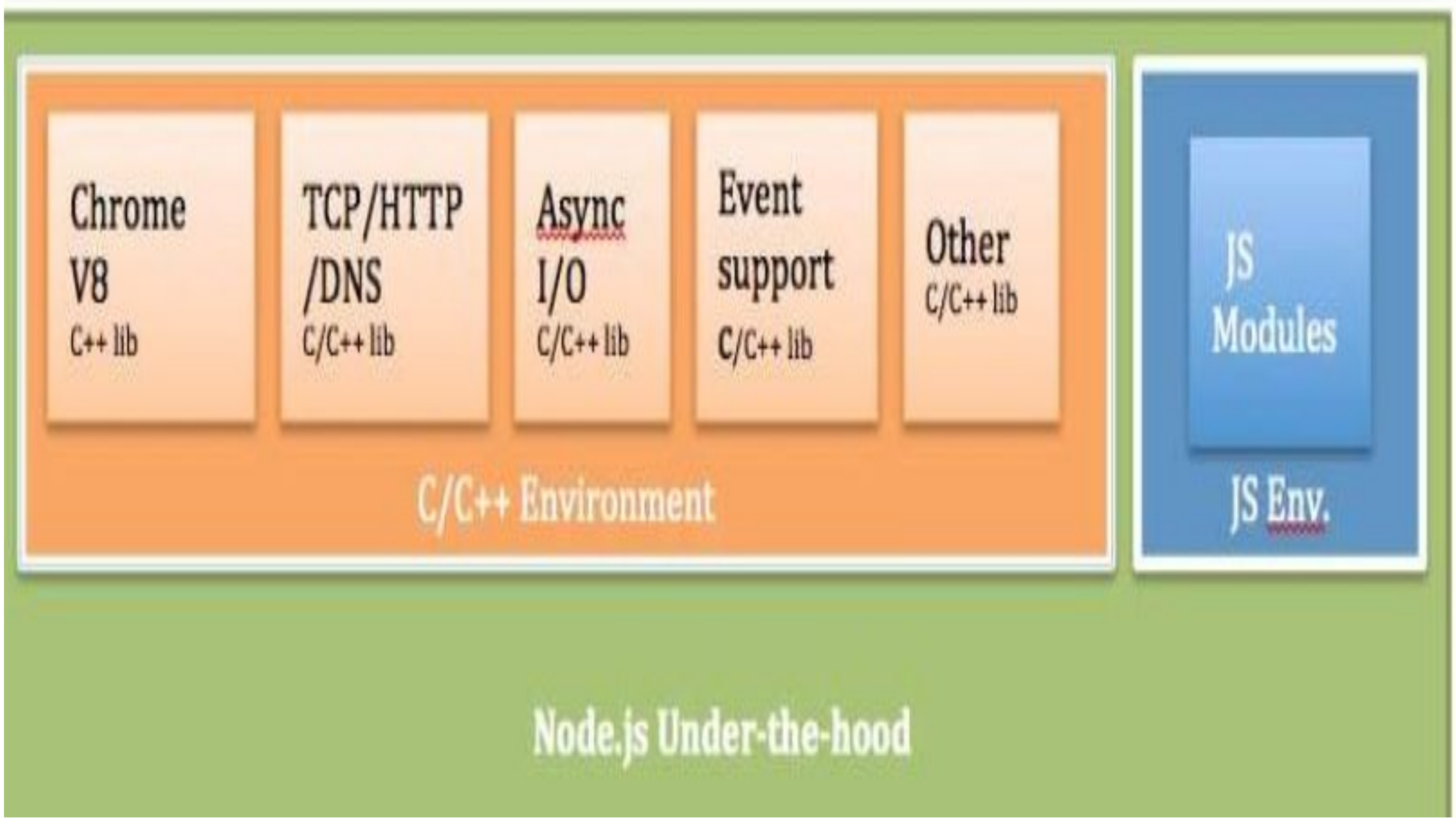
Features of node.js ?

1. **Extremely fast:** Node.js is built on Google Chrome's V8 JavaScript Engine, so its library is very fast in code execution.
2. **I/O is Asynchronous and Event Driven:** All APIs of Node.js library are asynchronous i.e. non-blocking. So a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call. It is also a reason that it is very fast.
3. **Single threaded:** Node.js follows a single threaded model with event looping.
4. **Highly Scalable:** Node.js is highly scalable because event mechanism helps the server to respond in a non-blocking way.
5. **No buffering:** Node.js cuts down the overall processing time while uploading audio and video files. Node.js applications never buffer any data. These applications simply output the data in chunks.
6. **Open source:** Node.js has an open source community which has produced many excellent modules to add additional capabilities to Node.js applications. License: Node.js is released under the MIT license.

Node.js Building Blocks



Node.js Under the Hood



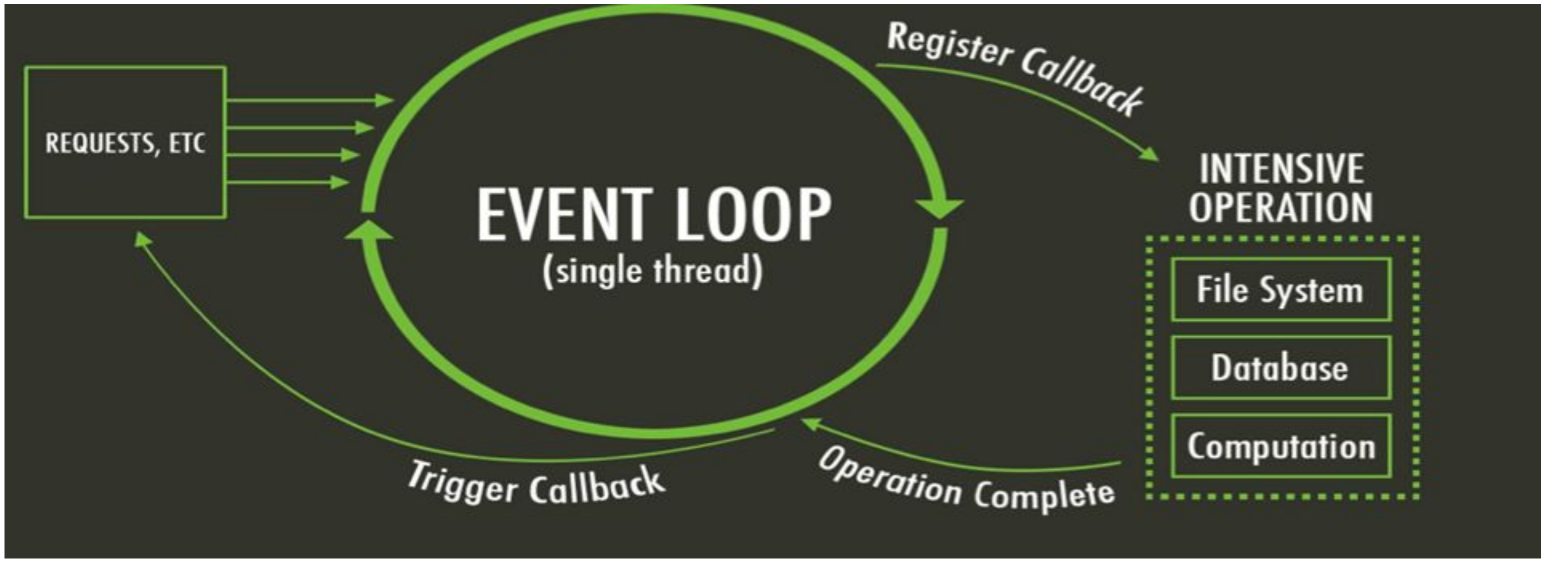
Standard JavaScript with

- Buffer
- C/C++ Addons
- Child Processes
- Cluster
- Console
- Crypto
- Debugger
- DNS
- Domain
- Events
- File System
- Globals
- HTTP
- HTTPS
- Modules
- Net
- OS
- Path
- Process
- Punycode
- Query Strings
- Readline
- REPL
- Stream
- String Decoder
- Timers
- TLS/SSL
- TTY
- UDP/Datagram
- URL
- Utilities
- VM
- ZLIB

... but without DOM manipulation

What is unique about Node.js?

1. Servers are normally thread based but Node.JS is “Event” based. Node.JS serves each request in a Evented loop that can handle simultaneous requests.
2. "Node is similar in design to and influenced by systems like Ruby's event machine or Python's twisted. Node takes the event model a bit further—it presents the event loop as a language construct instead of as a library."



Why node.js use event-based?

- In a normal process, the webserver while processing the request will have to wait for the IO operations and thus blocking the next request to be processed.

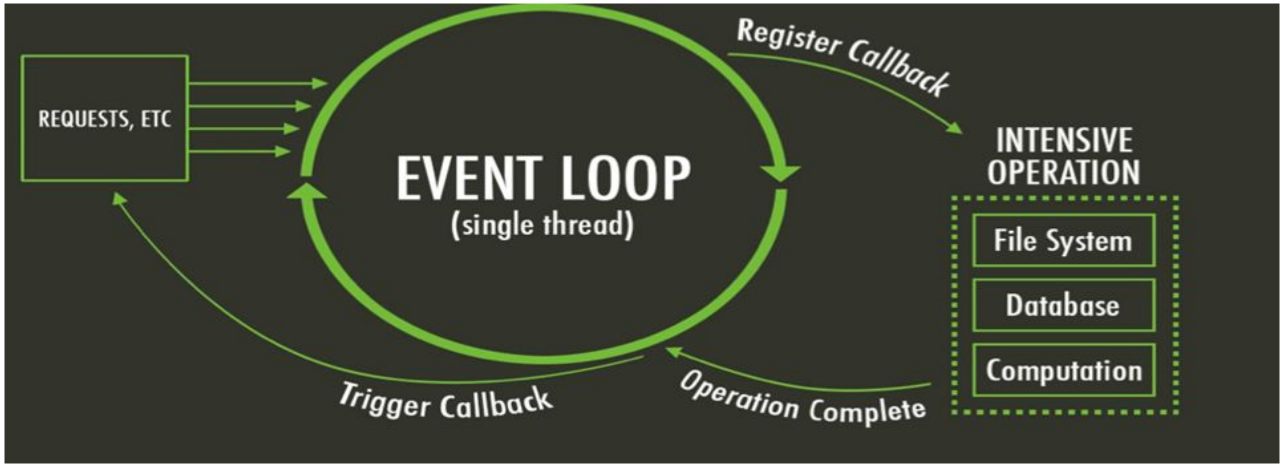
- Node.JS process each request as events, doesn't wait (non-blocking) for the IO operation to complete it can handle other request at the same time.

- When the IO operation of first request is completed it will call-back the server to complete the request.

- To avoid blocking, Node makes use of the event driven nature of JS by attaching callbacks to I/O requests (Internally, Node.js uses libuv for the event loop which in turn uses internal C++ thread pool to provide asynchronous I/O.)

- Scripts waiting on I/O waste no space because they get popped off the stack when their non-I/O related code finishes executing

Alleviates overhead of context switching

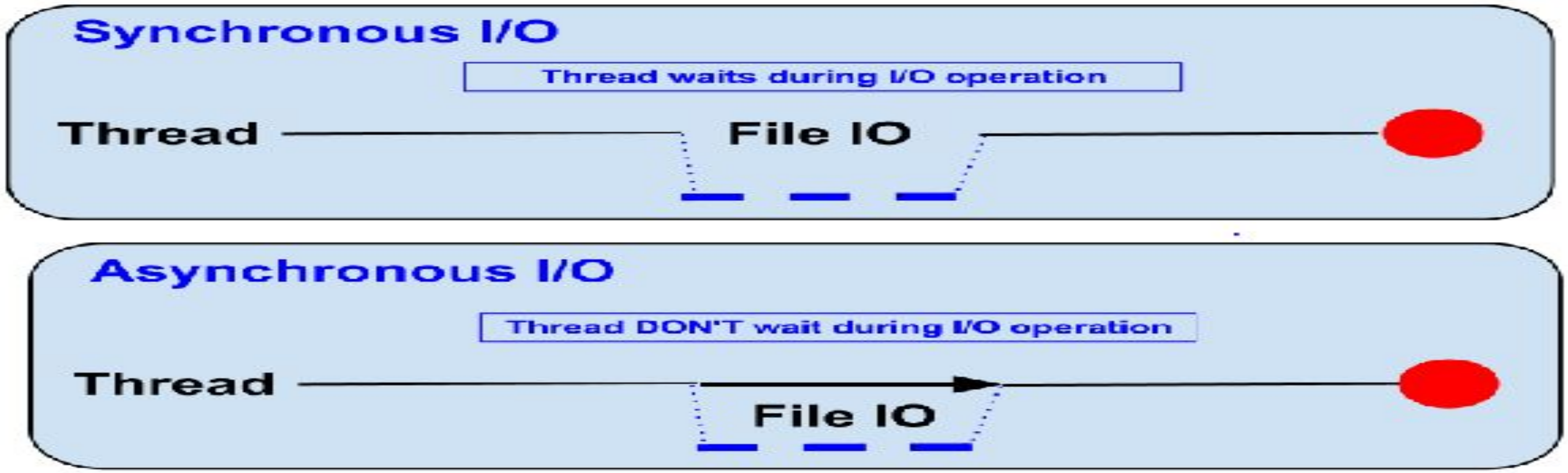


Threads VS Event-driven

Threads	Asynchronous Event-driven
Lock application / request with listener-workers threads	only one thread, which repeatedly fetches an event
Using incoming-request model	Using queue and then processes it
multithreaded server might block the request which might involve multiple events	manually saves state and then goes on to process the next event
Using context switching	no contention and no context switches
Using multithreading environments where listener and workers threads are used frequently to take an incoming-request lock	Using asynchronous I/O facilities (callbacks, not poll/select or O_NONBLOCK) environments

Blocking vs Non-Blocking……

Example :: Read data from file and show data


Synchronous I/O
Thread waits during I/O operation
Thread
File IO
Asynchronous I/O
Thread DON'T wait during I/O operation
Thread
File IO

Blocking…..

- Read data from file
- Show data
- Do other tasks

```
var data = fs.readFileSync( “test.txt” );
console.log( data );
console.log( “Do other tasks” );
```

Non-Blocking……

- Read data from file

  When read data completed, show data
- Do other tasks

```
fs.readFile( “test.txt”, function( err, data ) {
console.log(data);
});
console.log( “Do other tasks” );
```

Callback

● Callback is an asynchronous equivalent for a function. A callback function is called at the completion of a given task. Node makes heavy use of callbacks. All the APIs of Node are written in such a way that they support callbacks.

For example, a function to read a file may start reading file and return the control to the execution environment immediately so that the next instruction can be executed. Once file I/O is complete, it will call the callback function while passing the callback function, the content of the file as a parameter. So there is no blocking or wait for File I/O. This makes Node.js highly scalable, as it can process a high number of requests without waiting for any function to return results.

Blocking Code

```
●var fs = require("fs");
var data = fs.readFileSync('input.txt');

console.log(data.toString());
console.log("end");
```

O/P

```
FILE contents
end
```

The program blocks until it reads the file and then only it proceeds to end the program.

Non Blocking Code

```
●var fs = require("fs");
fs.readFile('input.txt', function (err, data) {
   if (err) return console.error(err);
   console.log(data.toString());
});
console.log("end");
```

O/P

```
end
FILE contents
```

Program does not wait for file reading and proceeds to print "end" and at the same time, the program without blocking continues reading the file.

Non-blocking programs do not execute in sequence. In case a program needs to use any data to be processed, it should be kept within the same block to make it sequential execution.

Node.js for….

- Create a js file named main.js on your machine (Windows or Linux) having the following code.

```
/* Hello, World! program in node.js */
console.log("Hello, World!")
```

Now execute main.js file using Node.js interpreter to see the result −

```
$ node main.js
```

```
- Hello, World!
```

REPL Terminal

● REPL stands for Read Eval Print Loop

It invokes a command prompt environment like a Linux shell where a command is entered and the system responds with an output in an interactive mode. Node.js or Node comes bundled with a REPL environment. It performs the following tasks −

Read − Reads user's input, parses the input into JavaScript data-structure, and stores in memory.

Eval − Takes and evaluates the data structure.

Print − Prints the result.

Loop − Loops the above command until the user presses ctrl-c twice.

The REPL feature of Node is very useful in experimenting with Node.js codes and to debug JavaScript codes.

Just type node on command prompt.

```
$ node
```

Node.js Basics

● Node.js supports JavaScript. So, JavaScript syntax on Node.js is similar to the browser's JavaScript syntax.

Node.js includes following primitive types:

- String
- Number
- Boolean
- Undefined
- Null
- RegExp
- Object

- JavaScript in Node.js supports loose typing like the browser's JavaScript. Use var keyword to declare a variable of any type.
- Object literal syntax is same as browser's JavaScript.

```
var obj = {
    username: 'user1',
    password: 'Node.js'
}
```

Functions and object

```
function Display(x) {
    console.log(x);
}
Display(100);
----
var person = {
  firstName: "Fname",
  lastName : "Lname",
  id       : 5,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
var p1=Object.create(person);
p1.fullName();
```

Errors

The Node.js applications generally face four types of errors:

1. Standard JavaScript errors i.e. <EvalError>, <SyntaxError>, <RangeError>, <ReferenceError>, <TypeError>, <URIError> etc.
2. System errors
3. User-specified errors
4. Assertion errors

```
// Throws with a ReferenceError because b is undefined
    try {
      const a = 1;
      const c = a + b;
    } catch (err) {
      console.log(err);
    }
```

Errors

```
const fs = require('fs');
function nodeStyleCallback(err, data) {
 if (err) {
   console.error('There was an error', err);
   return;
 }
 console.log(data);
}
fs.readFile('/some/file/that/does-not-exist', nodeStyleCallback);
fs.readFile('/some/file/that/does-exist', nodeStyleCallback);
```

Debug node.js application

You can debug Node.js application using various tools including following:

1. Core Node.js debugger

Node.js provides built-in non-graphic debugging tool that can be used on all platforms. It provides different commands for debugging Node.js application.

```
var fs = require('fs');
fs.readFile('test.txt', 'utf8', function (err, data) {
   debugger;
    if (err) throw err;
    console.log(data);
});  // run with node debug p1.js
```

Write debugger in your JavaScript code where you want debugger to stop. For example, we want to check the "data" parameter in the above example.

2. Node Inspector   npm install -g node-inspector

```
C:/> node-inspector --web-port=5500
```

3. Built-in debugger in IDEs

Modules in node.js

- Modules are JavaScript libraries.

A set of functions you want to include in your application.

There are two typesof modules:

1) Built-in

2) External

Built-in Modules

Node.js has a set of built-in modules which you can use without any further installation.

To include a module, use the require() function with the name of the module:

```
var http = require('http');
```

Now your application has access to the HTTP module, and is able to create a server.

Built-in http Module

● The HTTP module can create an HTTP server that listens to server ports and gives a response back to the client.

Use the createServer() method to create an HTTP server

The function passed into the http.createServer() method, will be executed when someone tries to access the computer on port 8080.

If the response from the HTTP server is supposed to be displayed as HTML, you should include an HTTP header with the correct content type:

```
res.writeHead(200, {'Content-Type': 'text/html'});
```

The function passed into the http.createServer() has a req argument that represents the request from the client, as an object (http.IncomingMessage object).

This object has a property called "url" which holds the part of the url that comes after the domain name.

http://localhost:8080/**emp**

Creating Node.js Application

- Step 1 - Import Required Module

We use the require directive to load the http module and store the returned HTTP instance into an http variable as follows −

```
var http = require("http");
```

Creating Node.js Application

- Step 2 - Create Server

We use the created http instance and call http.createServer() method to create a server instance and then we bind it at port 8081 using the listen method associated with the server instance. Pass it a function with parameters request and response. Write the sample implementation to always return "Hello World".

Step 3 - Testing Request & Response

Now execute the main.js to start the server as follows

```
$ node main.js
```

Verify the Output. Server has started.

```
Server running at http://127.0.0.1:8081/
```

Utility modules

- os Provides basic operating-system related utility functions.
- path Provides utilities for handling and transforming file paths.
- net Provides both servers and clients as streams. Acts as a network wrapper.
- dns Provides functions to do actual DNS lookup as well as to use underlying operating system name resolution functionalities.
- domain Provides ways to handle multiple different I/O operations as a single group.
- crypto Provides cryptographic functionality that includes a set of wrappers for open SSL's hash HMAC, cipher, decipher, sign and verify functions.
- tls TLS stands for Transport Layer Security. It is the successor to Secure Sockets Layer (SSL). TLS along with SSL is used for cryptographic protocols to secure communication over the web. TLS uses public-key cryptography to encrypt messages. It encrypts communication generally on the TCP layer.

•querystring To handle URL query strings

•readline To handle readable streams one line at the time

•stream To handle streaming data

•string_decoder To decode buffer objects into strings

•timers To execute a function after a given number of milliseconds

Utility modules

- querystring To handle URL query strings
- readline To handle readable streams one line at the time
- stream To handle streaming data
- string_decoder To decode buffer objects into strings
- timers To execute a function after a given number of milliseconds
- tty Provides classes used by a text terminal
- url To parse URL strings
- util To access utility functions
- v8 To access information about V8 (the JavaScript engine)
- zlib To compress or decompress files
- buffer To handle binary data

Node.js global objects

Node.js global objects are global in nature and they are available in all modules. We do not need to include these objects in our application, rather we can use them directly. These objects are modules, functions, strings and object itself.

1 **__filename** represents the filename of the code being executed. This is the resolved absolute path of this code file.

2 **__dirname** represents the name of the directory that the currently executing script resides in.

3 **setTimeout(cb, ms)** global function is used to run callback cb after at least ms milliseconds. The actual delay depends on external factors like OS timer granularity and system load. A timer cannot span more than 24.8 days. eg

```
function printHello() {
   console.log( "Hello, World!");
}
// Now call above function after 2 seconds
setTimeout(printHello, 2000);
```

Node.js global objects

4 **clearTimeout(t)** global function is used to stop a timer that was previously created with setTimeout(). Here t is the timer returned by the setTimeout() function.

```
function printHello() {
   console.log( "Hello, World!");
}
var t = setTimeout(printHello, 2000);
clearTimeout(t);
```

5. **setInterval(cb, ms)** global function is used to run callback cb repeatedly after at least ms milliseconds. The actual delay depends on external factors like OS timer granularity and system load. A timer cannot span more than 24.8 days.

This function returns an opaque value that represents the timer which can be used to clear the timer using the function clearInterval(t).

```
function printHello() {
   console.log( "Hello, World!");
}
setInterval(printHello, 2000);
```

Node.js global objects

6 console

7 process

8 buffer

Node.js console

console is a global object and is used to print different levels of messages to stdout and stderr. There are built-in methods to be used for printing informational, warning, and error messages.

It is used in synchronous way when the destination is a file or a terminal and in asynchronous way when the destination is a pipe.

There are three console methods that are used to write any node.js stream:

1. **console.log()**

    console.log('Hello %s', 'Students');   // can use format specifier

Prints to stdout with newline. This function can take multiple arguments like printf()..

2. **console.error() // to render error msg on console**

    console.error(new Error('Error Check Data.'));

3. **console.warn()**

4. **console.info() Alias of console.log()**

5. **console.dir(obj[, options])**

Uses util.inspect on obj and prints resulting string to stdout

Node.js console

6. console.time(label)

Mark a time.

7. console.timeEnd(label)

Finish timer, record output.

```
console.info("Program Started");
var counter = 10;
console.log("Counter: %d", counter);
console.time("Getting data");
console.timeEnd('Getting data');
console.info("Program Ended")
```

Node.js process object

The process object is a global object and can be accessed from anywhere. The process object is an instance of EventEmitter and emits the following events −

1 exit

Emitted when the process is about to exit. There is no way to prevent the exiting of the event loop at this point, and once all exit listeners have finished running, the process will exit.

2 beforeExit

This event is emitted when node empties its event loop and has nothing else to schedule. Normally, the node exits when there is no work scheduled, but a listener for 'beforeExit' can make asynchronous calls, and cause the node to continue.

3 uncaughtException

Emitted when an exception bubbles all the way back to the event loop. If a listener is added for this exception, the default action (which is to print a stack trace and exit) will not occur.

4 Signal Events

Emitted when the processes receives a signal such as SIGINT, SIGHUP, etc.

exit codes

Node normally exits with a 0 status code when no more async operations are pending. There are other exit codes which are described below −

1 Uncaught Fatal Exception

There was an uncaught exception, and it was not handled by a domain or an uncaughtException event handler.

2 Unused

reserved by Bash for built in misuse.

3 Internal JavaScript Parse Error

The JavaScript source code internal in Node's bootstrapping process caused a parse error. This is extremely rare, and generally can only happen during the development of Node itself.

4 Internal JavaScript Evaluation Failure

The JavaScript source code internal in Node's bootstrapping process failed to return a function value when evaluated. This is extremely rare, and generally can only happen during the development of Node itself.

5 Fatal Error

There was a fatal unrecoverable error in V8. Typically, a message will be printed to stderr with the prefix FATAL ERROR.

exit codes

6 Non-function Internal Exception Handler

There was an uncaught exception, but the internal fatal exception handler function was somehow set to a non-function, and could not be called.

7 Internal Exception Handler Run-Time Failure

There was an uncaught exception, and the internal fatal exception handler function itself threw an error while attempting to handle it.

8 Unused

9 Invalid Argument

Either an unknown option was specified, or an option requiring a value was provided without a value.

10 Internal JavaScript Run-Time Failure

The JavaScript source code internal in Node's bootstrapping process threw an error when the bootstrapping function was called. This is extremely rare, and generally can only happen during the development of Node itself.

11 Invalid Debug Argument

The --debug and/or --debug-brk options were set, but an invalid port number was chosen.

12 Signal Exits

If Node receives a fatal signal such as SIGKILL or SIGHUP, then its exit code will be 128 plus the value of the signal code. This is a standard Unix practice, since exit codes are defined to be 7-bit integers, and signal exits set the high-order bit, and then contain the value of the signal code.

Process properties

Process provides many useful properties to get better control over system interactions.

1 stdout

A Writable Stream to stdout.

2 stderr

A Writable Stream to stderr.

3 stdin

A Writable Stream to stdin.

4 argv

An array containing the command line arguments. The first element will be 'node', the second element will be the name of the JavaScript file. The next elements will be any additional command line arguments.

5 execPath

This is the absolute pathname of the executable that started the process.

6 env

An object containing the user environment.

Process properties

8 exitCode

A number which will be the process exit code, when the process either exits gracefully, or is exited via process.exit() without specifying a code.

9 version A compiled-in property that exposes NODE_VERSION.

10 versions

A property exposing the version strings of node and its dependencies.

11 config

An Object containing the JavaScript representation of the configure options that were used to compile the current node executable. This is the same as the "config.gypi" file that was produced when running the ./configure script.

12 pid The PID of the process.

13 title Getter/setter to set what is displayed in 'ps'.

14 arch What processor architecture you're running on: 'arm', 'ia32', or 'x64'.

15 platform

What platform you're running on: 'darwin', 'freebsd', 'linux', 'sunos' or 'win32'

16 mainModule

Alternate way to retrieve require.main. The difference is that if the main module changes at runtime, require.main might still refer to the original main module in modules that were required before the change occurred. Generally it's safe to assume that the two refer to the same module.

Process properties

```
// Printing to console
process.stdout.write("Hello World!" + "\n");
// Reading passed parameter
process.argv.forEach(function(val, index, array) {
  console.log(index + ': ' + val);
});
// Getting executable path
console.log(process.execPath);
// Platform Information
console.log(process.platform);
```

Process methods

1 abort() It causes the node to emit an abort, exit and generate a core file.

2 chdir(directory)

Changes the current working directory of the process or throws an exception if that fails.

3 cwd()

Returns the current working directory of the process.

4 exit([code])

Ends the process with the specified code. If omitted, exit uses the 'success' code 0.

5 getgid()

Gets the group identity of the process. This is the numerical group id, not the group name.This function is available only POSIX platforms (i.e. not Windows, Android).

6 setgid(id)

Sets the group identity of the process. (See setgid(2)). It accepts either a numerical ID or a groupname string. If a groupname is specified, this method blocks while resolving it to a numerical ID.This function is available only POSIX platforms (i.e. not Windows, Android).

7 getuid()

Gets the user identity of the process. This is the numerical id, not the username.This function is only available on POSIX platforms (i.e. not Windows, Android).

8 setuid(id)

Sets the user identity of the process (See setgid(2)). It accepts either a numerical ID or a username string. If a username is specified, this method blocks while resolving it to a numerical ID.This function is available only POSIX platforms (i.e. not Windows, Android).

Process methods

9 getgroups()

Returns an array with the supplementary group IDs. POSIX leaves it unspecified if the effective group ID is included, but node.js ensures it always is. This function is available only on POSIX platforms (i.e. not Windows, Android).

10 setgroups(groups)

Sets the supplementary group IDs. This is a privileged operation, which implies that you have to be at the root or have the CAP_SETGID capability. This function is available only on POSIX platforms (i.e. not Windows, Android).

11 initgroups(user, extra_group)

Reads /etc/group and initializes the group access list, using all the groups of which the user is a member. This is a privileged operation, which implies that you have to be at the root or have the CAP_SETGID capability. This function is available only on POSIX platforms (i.e. not Windows, Android).

12 kill(pid[, signal])

Send a signal to a process. pid is the process id and signal is the string describing the signal to send. Signal names are strings like 'SIGINT' or 'SIGHUP'. If omitted, the signal will be 'SIGTERM'.

13 memoryUsage()

Returns an object describing the memory usage of the Node process measured in bytes.

14 nextTick(callback)

Once the current event loop turn runs to completion, call the callback function.

Process methods

15 umask([mask])

Sets or reads the process's file mode creation mask. Child processes inherit the mask from the parent process. Returns the old mask if mask argument is given, otherwise returns the current mask.

16 uptime()

Number of seconds Node has been running.

17 hrtime()

Returns the current high-resolution real time in a [seconds, nanoseconds] tuple Array. It is relative to an arbitrary time in the past. It is not related to the time of day and therefore not subject to clock drift. The primary use is for measuring performance between intervals.

```
// Print the current directory
console.log('Current directory: ' + process.cwd());
// Print the process version
console.log('Current version: ' + process.version);
// Print the memory usage
console.log(process.memoryUsage());
```

Event Driven Programming with Event Loop

● Node.js is a single-threaded application, but it can support concurrency via the concept of event and callbacks.

Every API of Node.js is asynchronous and being single-threaded, they use async function calls to maintain concurrency.

Node.js uses events heavily and it is also one of the reasons why Node.js is pretty fast compared to other similar technologies. As soon as Node starts its server, it simply initiates its variables, declares functions and then simply waits for the event to occur.

Node uses **observer pattern**. Node thread keeps an event loop and whenever a task gets completed, it fires the corresponding event which signals the event-listener function to execute.

In an event-driven application, there is generally a main loop that listens for events, and then triggers a callback function when one of those events is detected.

Event Driven Programming

- Why Node JS is faster?

Compared to other technologies, Node JS applications provide very high performance because of Event-Driven Programming model.

- What is EventEmitter?

Node JS “events” module has one and only one class to handle events: EventEmitter class. It contains all required functions to take care of generating events.

- Who is responsible to generate events in Node JS Application ?

EventEmitter class is responsible to generate events. Generating events is also known as Emitting. That’s why this class name is EventEmitter as it emits events in Node JS Platform.

After generating events, it places all events into Event Queue. Then Event Loop picks-up events one by one from Event Queue and process them accordingly.

Event Driven Programming

- Who is responsible to handle events in Node JS Application ?

Node JS Platform uses the following two components to handle events.

1. EventEmitter class
2. Java Script Callback functions

EventEmitter class is responsible to generate events and Java Script Callback functions are responsible to handle them.

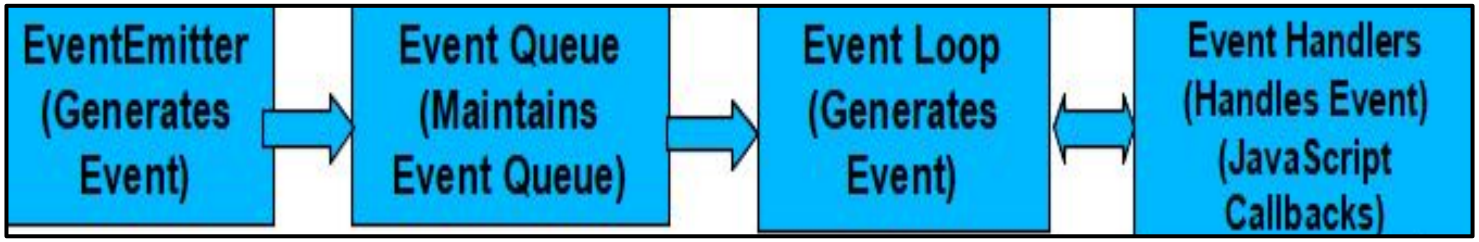
- When Node JS application starts or ends an operation, EventEmitter class generates events and places them into Event Queue.

- Event Queue maintains a Queue of Events.

- Event Loop continuously waits for new events in Event Queue. When it finds events in Event Queue, it pulls them and try to process them. If they require IO Blocking operations or long waiting tasks, then assigns respective Event Handlers to handle them.

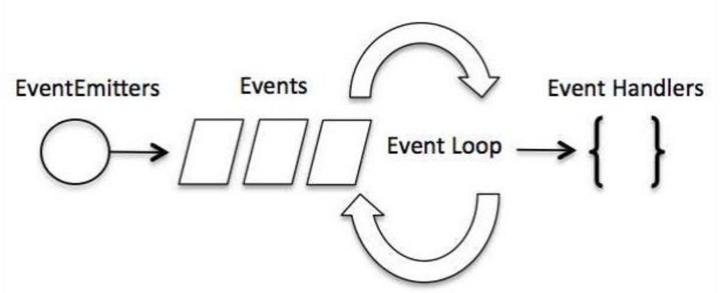
- Event Handlers are JavaScript Asynchronous Callback Functions. They are responsible to handle events and return results to Event Loop.

- Event Loop will prepare results and send them back to the Client.


EventEmitter (Generates Event)
Event Queue (Maintains Event Queue)
Event Loop (Generates Event)
Event Handlers (Handles Event) (JavaScript Callbacks)

Event Driven Programming with Event Loop

● Although events look quite similar to callbacks, the difference lies in the fact that callback functions are called when an asynchronous function returns its result, whereas event handling works on the observer pattern. The functions that listen to events act as Observers. Whenever an event gets fired, its listener function starts executing. Node.js has multiple in-built events available through events module and EventEmitter class which are used to bind events and event-listeners as follows −



```
// Import events module
var events = require('events');
// Create an eventEmitter object
var eventEmitter = new events.EventEmitter();
// Bind event and event  handler as follows
eventEmitter.on('eventName', eventHandler);
// Fire an event
eventEmitter.emit('eventName');
```

Event Emitter

- EventEmitter class lies in the events module.

When an EventEmitter instance faces any error, it emits an 'error' event. When a new listener is added, 'newListener' event is fired and when a listener is removed, 'removeListener' event is fired.

EventEmitter provides multiple methods like **on** and **emit**.

on property is used to bind a function with the event

emit is used to fire an event.

Event Emitter Methods

1. addListener(event, listener)

Adds a listener at the end of the listeners array for the specified event. No checks are made to see if the listener has already been added. Multiple calls passing the same combination of event and listener will result in the listener being added multiple times. Returns emitter, so calls can be chained.

2. on(event, listener)

alias of addListener

3. once(event, listener)

Adds a one time listener to the event. This listener is invoked only the next time the event is fired, after which it is removed. Returns emitter, so calls can be chained.

4. removeListener(event, listener)

Removes a listener from the listener array for the specified event. Caution − It changes the array indices in the listener array behind the listener. removeListener will remove, at most, one instance of a listener from the listener array. If any single listener has been added multiple times to the listener array for the specified event, then removeListener must be called multiple times to remove each instance. Returns emitter, so calls can be chained.

Event Emitter Methods

5 removeAllListeners([event])

Removes all listeners, or those of the specified event. It's not a good idea to remove listeners that were added elsewhere in the code, especially when it's on an emitter that you didn't create (e.g. sockets or file streams). Returns emitter, so calls can be chained.

6 setMaxListeners(n)

By default, EventEmitters will print a warning if more than 10 listeners are added for a particular event. This is a useful default which helps finding memory leaks. Obviously not all Emitters should be limited to 10. This function allows that to be increased. Set to zero for unlimited.

7 listeners(event)

Returns an array of listeners for the specified event.

8 emit(event, [arg1], [arg2], [...])

Execute each of the listeners in order with the supplied arguments. Returns true if the event had listeners, false otherwise.

Event Emitter Class Methods

1. listenerCount(emitter, event)

Returns the number of listeners for a given event.

Event Emitter Events

1. newListener

   event − String: the event name

   listener − Function: the event handler function

This event is emitted any time a listener is added. When this event is triggered, the listener may not yet have been added to the array of listeners for the event.

2. removeListener

   event − String The event name

   listener − Function The event handler function

This event is emitted any time someone removes a listener. When this event is triggered, the listener may not yet have been removed from the array of listeners for the event.

NPM

NPM

● Node Package Manager (NPM) provides two main functionalities −

Online repositories for node.js packages/modules

Command line utility to install Node.js packages, do version management and dependency management of Node.js packages.

https://www.npmjs.com/

A package in Node.js contains all the files you need for a module.

Modules are JavaScript libraries you can include in your project.

NPM comes bundled with Node.js installation.

To know version: −

```
$ npm --version
```

Install modules using NPM

- There is a simple syntax to install any Node.js module −

```
$ npm install <Module Name>
```

For example, following is the command to install a famous Node.js web framework module called express −

```
$ npm install express
```

Now you can use this module in your js file as following −

```
var express = require('express');
```

Global vs Local installation

- By default, NPM installs any dependency in the local mode. i.e. to the package installation in node_modules directory lying in the folder where Node application is present.

Locally deployed packages are accessible via require() method. For example, when we installed express module, it created node_modules directory in the current directory where it installed the express module.

```
$ ls -l
total 0
drwxr-xr-x 3 root root 20 Mar 17 02:23 node_modules
```

Alternatively, use **npm ls** command to list down all the locally installed modules.

Global vs Local installation

● Globally installed packages/dependencies are stored in system directory. Such dependencies can be used in CLI (Command Line Interface) function of any node.js.

```
$ npm install express -g
```

To check all the modules installed globally −

```
$ npm ls -g
```

Save dependency in package.json for future install:

```
$ npm i express --save
```

Save dev dependency in package.json (not needed in production):

```
$ npm i express -D (OR npm I express --save-dev)
```

Global vs Local installation

● Uninstalling a Module

```
$ npm uninstall express
```

Once NPM uninstalls the package, you can verify it by looking at the content of /node_modules/ directory or type the following command −

```
$ npm ls
```

Updating a Module

Update package.json and change the version of the dependency to be updated and run the following command.

```
$ npm update express
```

Search a package name using NPM.

```
$ npm search express
```

File package.json…..

package.json is present in the root directory of any Node application/module and is used to define the properties of a package.

Project information

- Name
- Version
- Dependencies
- Licence
- Main file

Etc...

```
{
  "name": "node-js-getting-started",
  "version": "0.2.5",
  "description": "A sample Node.js app using Express 4",
  "engines": {
    "node": "5.9.1"
  },
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "dependencies": {
    "body-parser": "^1.16.1",
    "cookie-parser": "^1.4.3",
    "cool-ascii-faces": "1.3.4",
    "ejs": "2.4.1",
    "express": "^4.13.3",
    "express-session": "^1.15.1",
    "mongodb": "^2.2.24",
    "multer": "^1.3.0",
    "pg": "4.x",
    "pug": "^2.0.0-beta11"
  },
  "repository": {
    "type": "git",
    "url": "https://github.com/heroku/node-js-getting-started"
  },
  "keywords": [
    "node",
    "heroku",
    "express"
  ],
  "license": "MIT"
}
```

Execute scripts with npm

1. npm init

2. npm start (by default starts server.js in current directory, else start script of package.json)

3. npm run say-hello

4. npm test

5 npm install

Package.json

```
"scripts": {

        "start": "node events2.js",

        "say-hello": "echo 'Hello World'",

        "awesome-npm": "npm run say-hello && echo 'echo NPM is awesome!'"

        "bash-hello": "bash hello.sh"
```

Execute scripts with npm

A Few Use Cases for NPM Scripts

There is a lot that you can do with NPM scripts. Some use cases are:

- Minification/Uglification of CSS/JavaScript
- Automating the build process
- Compressing images
- Automatically injecting changes with BrowserSync

Create your own Modules

- Create a module that returns the current date and time:

```
exports.myDateTime = function () {
  return Date();
};
```

Use the exports keyword to make properties and methods available outside the module file.

Save the code above in a file called "myfirstmodule.js"

Now you can include and use the module in any of your Node.js files.

Use the module "myfirstmodule" in a Node.js file:

```
var http = require('http');
var dt = require('./myfirstmodule');  // use of ./ = module is in current directory

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write("The date and time are currently: " + dt.myDateTime());
  res.end();
}).listen(8080);
```

Callback Hell

When we execute tasks which are dependent on each other, we wrap them into the callbacks of each function and hence caught into callback hell situation.

http://callbackhell.com/

To avoid callback hell

Follow one or combination of the following :

- Modularise your code. // Use separate functions and call in callbacks-lot of code
- Use event-driven programming. // file read, display event13
- Use promises //Q or Bluebird modules to avail the feature of promise.
- Use Async.js
- Use generators //convert asynchronous code to synchronous

Callbacks

```
const fs = require('fs')
fs.readFile('file.md', 'utf-8', function (err, content) {
  if (err) {    return console.log(err)  }
  console.log(content)
}) /--------------

var fs = require("fs");
var db = require('db');
var sendEmail = require('email');
fs.readFile('f1.js','utf8',function(err,data){
    if(err) {      console.log("Error",err);    }
    db.executeQuery('SELECT * FROM test',function(err,rows) {
      if(err) {        console.log("Error",err);       }
      sendEmail(rows,function(err,data) {
        if(err) {          console.log("Error",err);        }
        console.log("Operation done, i am in callback hell");
      });
    });
});
```

Promise

Promise represents the result of asynchronous function. Promises can be used to avoid nesting of callbacks. In Node.js, you can use Q or Bluebird modules to avail the feature of promise. (Apps: n1 -file file-promise,promise.js, node -db mp.js)

Core Promise – for files let promise =new Promise(function(resolve, reject) {})

```
function readFile(filename){
  return new Promise(function (success, fail){
    fs.readFile(filename, function (err, res){
      if (err)
        fail(err);
      else
        success(res);
    });
  });
}
readFile('./files/file1.txt')
  .then(function success(data)
  {
    console.log("File read="+data);
  })
  .catch(function fail(err)
  {
    console.log("ERROR="+err);
  });
```

Promises

fs.readFile(“file”,function(err,data){ }) // Callback

- Callback functions are used for Asynchronous events. But sometimes callback functions can become a nightmare when they start becoming nested, and the program starts to become long and complex.

- Promise is an enhancement to callbacks that looks towards alleviating these problems.

- "readFile" is a callback or asynchronous function which does some sort of processing.

- This time, when defining the callback, there is a value which is returned called a "promise."

  When a promise is returned, it can have 2 outputs. This is defined by the 'then clause'. Either the operation can be a success which is denoted by the 'onFulfilled' parameter. Or it can have an error which is denoted by the 'onRejected' parameter.

Promise

Core Promise - mongodb

```
//mongodb module by default provides promise support
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url,{ useNewUrlParser: true })
    .then(function(db){
       var dbo = db.db("mydb");
       database=db;
       var myobj = { name: "pqr", address: "abd" };
       return dbo.collection("users").insertOne(myobj);
    })
    .then(function(res){
       console.log("RES=="+res);
       database.close();
       process.exit(0)
    })
    .catch(function(err){console.log(err);});
```

Bluebord for Promise

```
npm i bluebird
---------
var Promise = require('bluebird');
var readFile = Promise.promisify(require("fs").readFile);
readFile("./files/file2.txt", "utf8").then(function(contents) {
   return JSON.parse(contents);
}).then(function(result) {
   console.log("The result of evaluating myfile.js", result);
}).catch(SyntaxError, function(e) {
   console.log("File had syntax error", e);
//Catch any other error
}).catch(function(e) {     console.log("Error reading file", e); });
```

Promise for DB Connect

```
let mongoose = require('mongoose');

const server = '127.0.0.1:27017'; // REPLACE WITH YOUR DB SERVER
const database = 'fcc-Mail'; // REPLACE WITH YOUR DB NAME
class Database {
  constructor() {
    this.connect-db()
  }

connect-db() {
     mongoose.connect(`mongodb://${server}/${database}`)
       .then() => {
         console.log('Database connection successful')
       })
       .catch(err => {
         console.error('Database connection error')
       })
  }
}
module.exports = new Database()
```

Async

Async functions are essentially a cleaner way to work with asynchronous code in JavaScript. Using async-await, you can simplify your callback or Promise based Node.js application with async functions.

- It is the newest way to write asynchronous code in JavaScript.
- It is non-blocking (just like callbacks and promises).
- Async/Await is created to simplify the process of working with and writing chained promises.
- An async function returns the Promise. If the function throws an error, the Promise will be automatically rejected, and if a function returns the value that means the Promise will be resolved.

Async

With promise:

```
function square(x) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(Math.pow(x, 2));
    }, 2000);
  });
}
square(10).then(data => {
  console.log(data);
});
```

turn above code into async/await function.

```
function square(x) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(Math.pow(x, 2));
    }, 2000);
  });
}
async function async_square(x)
{
  const value = await square(x);
  console.log(value);
}
async_square(10);
```

```
async function addAsync(x) {
  const a = await square(10);
  const b = await square(20);
  const c = await square(30);
  return x + a + b + c;
}
addAsync(10).then((sum) => {
  console.log(sum);
});
```

Async

Async/Await in AJAX Request.

node-fetch library or request

```
npm install node-fetch --save
```

For the ajax request, we can use the async-await function like the following.

```
// server.js

const fetch = require('node-fetch');

async function asyncajaxawait(x)
{
  const res = await fetch('https://api')
  const data = await res.json();
  console.log(data.name);
}

asyncajaxawait(10);
```

Global object in node

//With strict mode, you can not, use undeclared variables.
'use strict';

Global Object

<Object> The global namespace object.

In browsers, the top-level scope is the global scope. That means that in browsers if you're in the global scope var something will define a global variable.

In Node.js this is different. The top-level scope is not the global scope; var something inside an Node.js module will be local to that module.

We can access the global object in node using the global keyword:
console.log(global);

global.var1=”val1”;

```
(function(){
  console.log(this);
}());
```

Prints the global object and in use strict mode prints undefined

```
(function(){
  console.log(this);
}());
```

Node-static and nodemon

npm install node-static

- The node-static library is a convenience that takes care of the process of responding to requests for the static HTML and JavaScript files that will run on the client side in the browser.

```
const static = require('node-static');
const fileServer = new static.Server('./public');
var server = http.createServer(function(request, response) {
        request.addListener('end', function () {
        var _get = url.parse(request.url, true).query;
        fileServer.serve(request, response);
        }).resume();
}).listen(8080);
```

- nodemon This tool restarts our server as soon as we make a change in any of our files, otherwise we need to restart the server manually after each file modification.

npm install -g nodemon

- Once installed, instead of node server.js use **nodemon server.js**

Network Programming

A socket is one endpoint of a two way communication link between two programs running on the network. The socket mechanism provides a means of inter-process communication (IPC) by establishing named contact points between which the communication take place.

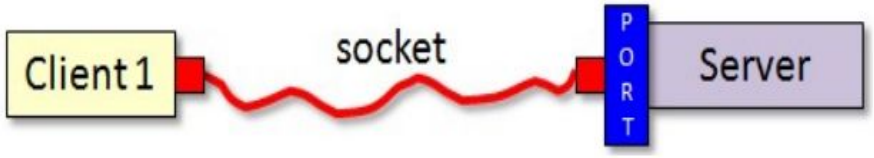
Unix Socket or Windows Socket Gives Linux, Mac, and Windows lightning fast communication and avoids the network card to reduce overhead and latency.

TCP Socket Gives the most reliable communication across the network. Can be used for local IPC as well, but is slower than #1's Unix Socket Implementation because TCP sockets go through the network card while Unix Sockets and Windows Sockets do not.

TLS Socket Configurable and secure network socket over SSL. Equivalent to https.

UDP Sockets Gives the fastest network communication. UDP is less reliable but much faster than TCP. It is best used for streaming non critical data like sound, video, or multiplayer game data as it can drop packets depending on network connectivity and other factors. UDP can be used for local IPC as well, but is slower than #1's Unix Socket or Windows Socket Implementation because UDP sockets go through the network card while Unix and Windows Sockets do not.

OS	**Supported Sockets**
Linux :	Unix, Posix, TCP, TLS, UDP
Mac:	Unix, Posix, TCP, TLS, UDP
Win:	Windows, TCP, TLS, UDP



Network Programming

Node.js **net** module is used to create both servers and clients. This module provides an asynchronous network wrapper.

Variants of sockets in Node:

- TCP
- UDP
- Websocket
- UNIX domain (node-ipc package)

There are two categories of TCP socket programs you can write:

- Server - listens for connections to it from clients and send data to the client
- Client - connects to a TCP server exchange data with it. The communication between client and server happens via sockets

<u>Npm packages:</u> **node-ipc, net, dgram, ws (wrappers: websocket, socket.io)**

Node-ipc: a nodejs module for local and remote Inter Process Communication with full support for Linux, Mac and Windows. It also supports all forms of socket communication from low level unix and windows sockets to UDP and secure TLS and TCP sockets.

Network Programming

net: Node's net module is a basic low level TCP and UDP interface. It allows you to make a TCP or UDP connection to some endpoint and to then send or receive data from that endpoint over that connection. These are raw TCP connections. You define the protocol, data format and all conventions used in the communication. All TCP does is deliver your data from one end to the other. **var net = require("net")**

<u>Network stack</u>

- socket.io, webSocket

- ws

- TCP

- UNIX/ WIN

ws - A **webSocket** is built (native) on top of TCP. It has it's own unique connection scheme that starts with an http connection with certain custom headers and then requests an "upgrade" to the webSocket protocol. If the server approves the upgrade, then the same TCP socket that the http connection started over is converted to the webSocket protocol. The webSocket protocol has it's own unique encryption and data format.

- **socket.io** is built on top of the webSocket protocol (meaning it uses the webSocket protocol for it's communication). Socket.io has it's own unique connection scheme (starts with http polling, then switches to a regular webSocket if permissible) and it has an additional data structure built on top of the webSocket data frame that defines a message name and a data packet and a few other housekeeping things.

- socket.io and webSocket are both supported from browser Javascript. A plain TCP or UDP connection is not supported from browser Javascript. So, if you were looking to communicate with a browser, you would not be using plain TCP

Websocket for Developers

- Web Socket is an independent TCP-based protocol, but it is designed to support any other protocol that would traditionally run only on top of a pure TCP connection.

- Web Socket is a transport layer on top of which any other protocol can run. The Web Socket API supports the ability to define sub-protocols: protocol libraries that can interpret specific protocols.

- Such protocols are **XMPP, STOMP, AMQP** etc. The developers no longer have to think in terms of the HTTP request-response paradigm.

- The only requirement on the browser-side is to run a JavaScript library that can interpret the Web Socket handshake, establish and maintain a Web Socket connection.

- On the server side, the industry standard is to use existing protocol libraries that run on top of TCP and leverage a Web Socket Gateway.

Websocket

Use cases are:

- Multiplayer online games
- Chat applications
- Live sports ticker
- Realtime updating social streams

Websocket

SERVER SIDE

- If you want to build your own WebSocket server.

- Websocket server can communicate with almost any kind of client.

- One of the most popular is socket.io, a Node.JS library that provides cross-browser fallbacks so you can use WebSockets in your applications.

Some other libraries in different languages include:

C++: libwebsockets

Erlang: Shirasu.ws

Java: Jetty

Node.JS: ws (Wrapper libraries: websocket, socket.io)

Ruby: em-websocket

Python: Tornado, pywebsocket

PHP: Ratchet, phpws

CLIENT SIDE

Browser(Client) Support for WebSockets

- WebSockets **(using WebSocket javascript object)** are supported in almost all modern web browsers.

- **Client may be :** webbrowser (javascript), mobile browser, flex, nodejs, java, php, mobile app, .net

Websocket

Web Sockets occupy a key role not only in the web but also in the mobile industry. The importance of Web Sockets is given below.

- Web Sockets as the name indicates, are related to the web. Web consists of a bunch of techniques for some browsers; it is a broad communication platform for vast number of devices, including desktop computers, laptops, tablets and smart phones.

- HTML5 app that utilizes Web Sockets will work on any HTML5 enabled web browser.

- Web socket is supported in the mainstream operating systems. All key players in the mobile industry provide Web Socket APIs in own native apps.

- Web sockets are said to be a full duplex communication. The approach of Web Sockets works well for certain categories of web application such as chat room, where the updates from client as well as server are shared simultaneously.

Websocket

The steps for establishing the connection of Web Socket are as follows −

- The client establishes a connection through a process known as Web Socket handshake.
- The process begins with the client sending a regular HTTP request to the server.
- An Upgrade header is requested. In this request, it informs the server that request is for Web Socket connection.
- Web Socket URLs use the ws scheme. They are also used for secure Web Socket connections, which are the equivalent to HTTPs.

<u>Sample request header for websocket protocol request</u>

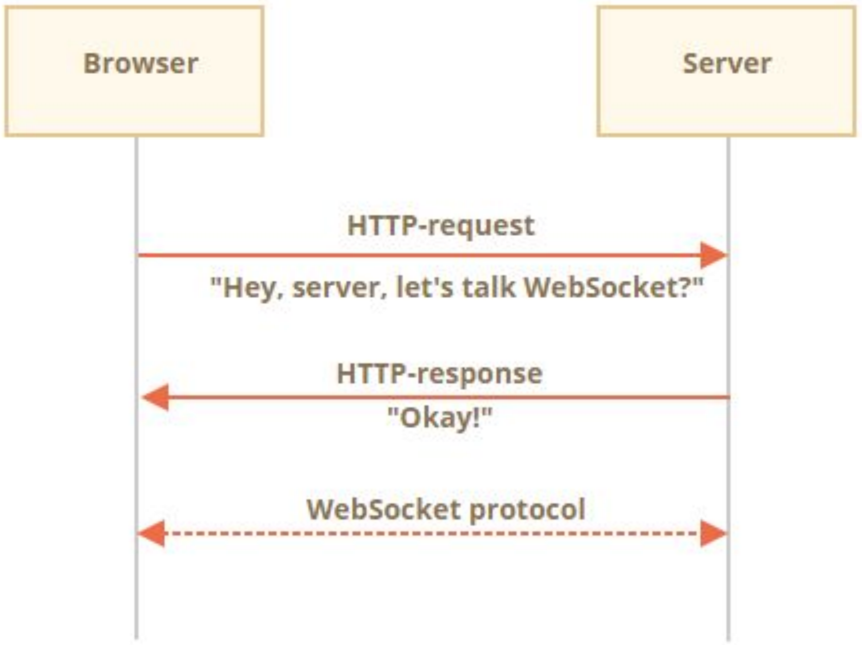
GET ws://websocket.example.com/ HTTP/1.1

Origin: http://www.abc.com

Connection: Upgrade

Host: websocket.abc.com

Upgrade: websocket



Websocket

Package: ws

- WebSockets are an alternative to HTTP communication in Web Applications.

- They offer a long lived, bidirectional communication channel between client and server.

- Once established, the channel is kept open, offering a very fast connection with low latency and overhead.

- WebSockets are supported by all modern browsers.

- HTTP is a very different protocol, and also a different way of communicate.

- HTTP is a request/response protocol: the server returns some data when the client requests it.

With WebSockets:

- The server can send a message to the client without the client explicitly requesting something
- The client and the server can talk to each other simultaneously

very little data overhead needs to be exchanged to send messages. This means a low latency communication.

Buffers

- Pure JavaScript is Unicode friendly, but it is not so for binary data.

- While dealing with TCP streams or the file system, it's necessary to handle octet streams.

- Node provides Buffer class which provides instances to store raw data similar to an array of integers but corresponds to a **raw memory allocation outside the V8 heap**.

- Buffer class is a global class that can be accessed in an application without importing the buffer module.

Creating Buffers

Node Buffer can be constructed in a variety of ways.

Method 1 Following is the syntax to create an uninitiated Buffer of 10 octets −

```
var buf = Buffer.alloc(10);
```

Method 2

Following is the syntax to create a Buffer from a given array −

```
var buf = Buffer.from([10, 20, 30, 40, 50]);
```

Method 3

Following is the syntax to create a Buffer from a given string and optionally encoding type −

```
var buf = Buffer.from("Node Buffer", "utf-8");
```

Though "utf8" is the default encoding, you can use any of the following encodings "ascii", "utf8", "utf16le", "ucs2", "base64" or "hex".

Buffers

Writing to Buffers

Syntax

buf.write(string[, offset][, length][, encoding])

Parameters

- string − This is the string data to be written to buffer.
- offset − This is the index of the buffer to start writing at. Default value is 0.
- length − This is the number of bytes to write. Defaults to buffer.length.
- encoding − Encoding to use. 'utf8' is the default encoding.

Return Value

This method returns the number of octets written. If there is not enough space in the buffer to fit the entire string, it will write a part of the string.

```
buf = Buffer.alloc(256);
len = buf.write("Node is fun!");
console.log("Octets written : "+  len);
```

op-

```
Octets written : 12
```

Buffers

Reading from Buffers

buf.toString([encoding][, start][, end])

encoding − Encoding to use. 'utf8' is the default encoding.

start − Beginning index to start reading, defaults to 0.

end − End index to end reading, defaults is complete buffer. (end-1)

Return Value

This method decodes and returns a string from buffer data encoded using the specified character set encoding.

```
buf = Buffer.allco(26);
for (var i = 0 ; i < 26 ; i++) {
  buf[i] = i + 97;
}
console.log( buf.toString('ascii'));       // outputs: abcdefghijklmnopqrstuvwxyz
console.log( buf.toString('ascii',0,5));   // outputs: abcde
console.log( buf.toString('utf8',0,5));    // outputs: abcde
console.log( buf.toString(undefined,0,5)); // encoding defaults to 'utf8', outputs abcde
```

Buffers

Convert Buffer to JSON

```
var buf = Buffer.from('Node buffer');
var json = buf.toJSON(buf);
console.log(json);
```

Concatenate Buffers

Buffer.concat(list[, totalLength])

- list − Array List of Buffer objects to be concatenated.
- totalLength − This is the total length of the buffers when concatenated.

```
var buffer1 = Buffer.from('buffer1 ');
var buffer2 = Buffer.from('buffer2');
var buffer3 = Buffer.concat([buffer1,buffer2]);
console.log("buffer3 content: " + buffer3.toString());
```

Buffer.isEncoding(encoding)

Buffer.isBuffer(obj)

Compare: buf.compare(otherBuffer);

Copy: buf.copy(targetBuffer[, targetStart][, sourceStart][, sourceEnd])

Slice: buf.slice([start][, end])

Length: buf.length;

byteLength(string,[encoding]): Gives the actual byte length of a string. encoding defaults to 'utf8'.

Streams

Streams are objects that let you read data from a source or write data to a destination in continuous fashion. In Node.js, there are four types of streams −

- Readable − Stream which is used for read operation.
- Writable − Stream which is used for write operation.
- Duplex − Stream which can be used for both read and write operation.
- Transform − A type of duplex stream where the output is computed based on input.

Each type of Stream is an EventEmitter instance and throws several events at different instance of times. For example, some of the commonly used events are −

- data − This event is fired when there is data is available to read.
- end − This event is fired when there is no more data to read.
- error − This event is fired when there is any error receiving or writing data.
- finish − This event is fired when all the data has been flushed to underlying system.

Streams

Streams work on the concept of buffer.

Using streams you read it piece by piece, processing its content without keeping it all in memory.

A buffer is a temporary memory that a stream takes to hold some data until it is consumed.

The Node.js stream module provides the foundation upon which all streaming APIs are built. All streams are instances of EventEmitter

Why streams

Memory efficiency: you don't need to load large amounts of data in memory before you are able to process it

Time efficiency: it takes way less time to start processing data as soon as you have it, rather than waiting till the whole data payload is available to start

V8

- V8 is an open source JavaScript engine developed by the Chromium project for the Google Chrome web browser. It is written in C++. Now a days, it is used in many projects such as Couchbase, MongoDB and Node.js.

- The Node.js V8 module represents interfaces and event specific to the version of V8. It provides methods to get information about heap memory through v8.getHeapStatistics() and v8.getHeapSpaceStatistics() methods.

- To use this module, you need to use **require('v8')**

- **v8.getHeapStatistics()** method returns statistics about heap such as total heap size, used heap size, heap size limit, total available size etc.

- **v8.getHeapSpaceStatistics()** returns statistics about heap space. It returns an array of 5 objects: new space, old space, code space, map space and large object space. Each object contains information about space name, space size, space used size, space available size and physical space size.

ALTERNATIVE to install - generator

```
npm install express-generator -g express helloapp
create : helloapp
create : helloapp/package.json
create : helloapp/app.js
create : helloapp/public
create : helloapp/public/images
create : helloapp/routes
create : helloapp/routes/index.js
create : helloapp/routes/users.js
create : helloapp/public/stylesheets
create : helloapp/public/stylesheets/style.css
create : helloapp/views create : helloapp/views/index.jade
create : helloapp/views/layout.jade
create : helloapp/views/error.jade
create : helloapp/bin
create : helloapp/bin/www

install dependencies:
$ cd helloapp && npm install
run the app:
$ DEBUG=helloapp:* npm start
create : helloapp/public/javascripts
```

Express – hello world code

- index.js have the code

```
var express = require('express

 var app = express()

 app.get('/', function (req, res)
      res.send('Hello World!')
 })

app.listen(3000, function () {
        console.log('Example app listening on port 3000!')
 })
```

This says requires module express

Calls function express to initialize object app

App object has various methods like get that responds to HTTP get request.
This code will be call the function specified when
a GET for the URI / is invoked

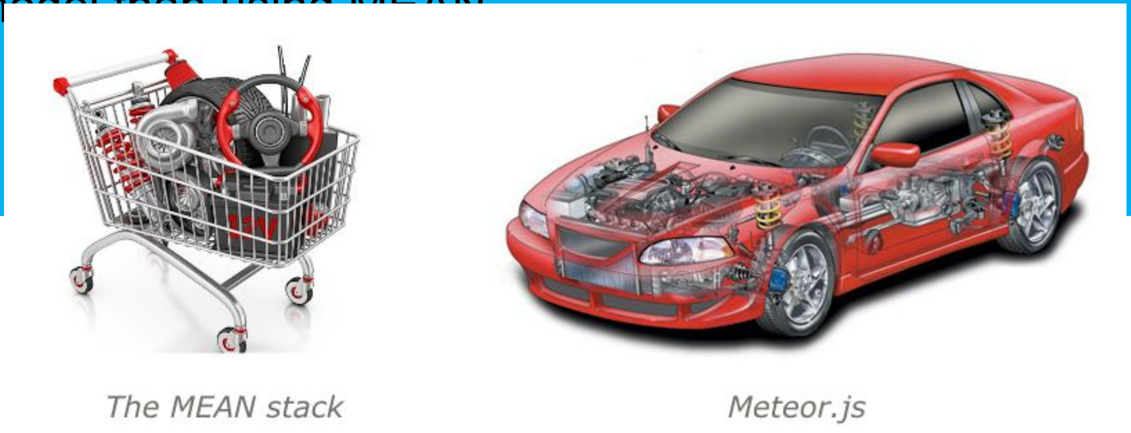
Sets up the HTTP server for listening port 3000

NEXT ------

- Go over class website to see how to use MongoDB and more on Express!!!

Once you are done, you will understand the basics of MEAN (without the A) and a start towards using NodeJS for web systems.

HOWEVER…..I might suggest next learning about Meteor – less callbacks, more
subscription m



NEXT ------

- Ws server, ws client, ws httpserver
- Built-in modules (self study)
- Websocket server client (cancel)
- Socketio server client (join/leave room)
- Chatbot
- Mysql,mongodb