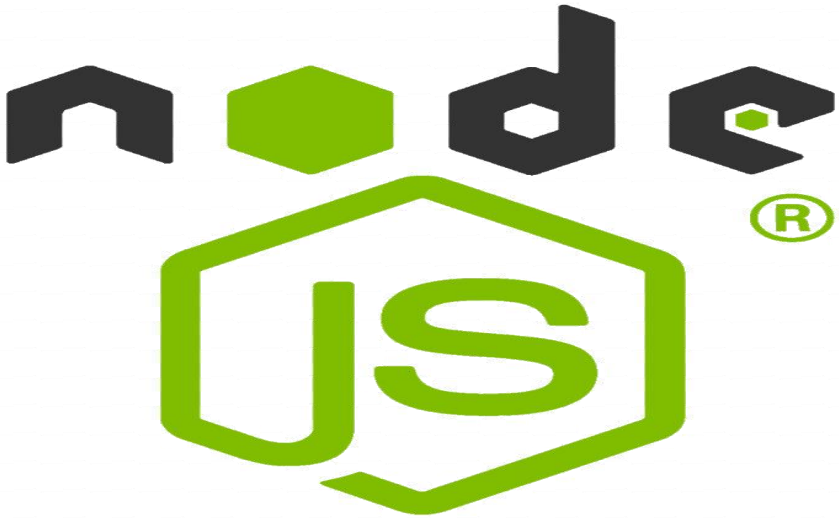
The Node.js Chronicles:

Node JS Top Interview Questions



Mastering Asynchronous Programming and Building Scalable Applications.

- Souvik Das

Unlocking the Potential of Server-Side JavaScript

Date of Publication: May 2024

Email: souda0303@gmail.com

Preface:

Welcome to this comprehensive guide on Node.js. This document is crafted to provide you with an in-depth understanding of Node.js, a powerful and efficient runtime that enables server-side scripting and scalable network applications.

Purpose of This Guide

The primary goal of this guide is to serve as an exhaustive resource for both beginners and seasoned developers. Whether you are new to Node.js or looking to deepen your existing knowledge, this guide offers detailed insights into its core concepts, advanced techniques, and best practices.

What You Will Learn

Foundations of Node.js: Understand the fundamental principles, including asynchronous programming, the event-driven architecture, and the V8 JavaScript engine.

Core Modules and APIs: Gain in-depth knowledge of Node.js built-in modules like http, fs, path, and many more, which are essential for creating efficient server-side applications.

Asynchronous Programming: Master the art of handling asynchronous operations using callbacks, Promises, and async/await.

Building and Scaling Applications: Learn how to build robust applications, from simple APIs to complex systems, and how to scale them effectively.

Error Handling and Debugging: Explore techniques for efficient error handling and debugging to create stable and reliable applications.

Security Best Practices: Discover how to secure your Node.js applications against common vulnerabilities and threats.

Performance Optimization: Delve into methods to optimize the performance of your Node.js applications for better responsiveness and lower latency.

Why Node.js?

Node.js has revolutionized server-side programming with its non-blocking, event-driven architecture, which allows for the handling of many connections simultaneously. This makes it ideal for real-time applications such as chat applications, gaming servers, and live data feeds. Furthermore, the large ecosystem of modules available through npm (Node Package Manager) simplifies development and accelerates productivity.

How to Use This Guide

This guide is structured to build your knowledge step-by-step, from basic concepts to advanced techniques. You can read it sequentially to gain a comprehensive understanding or jump to specific sections as needed. Each chapter includes practical examples and exercises to reinforce learning.

Community and Further Learning

The Node.js community is vibrant and continually evolving. Engaging with this community through forums, contributing to open-source projects, and following the latest developments will further enhance your skills and keep you updated with new advancements.

Embark on this journey to master Node.js and unlock the potential to build scalable, high-performance applications.

Contents:

Q17. How we can read file in synchronous way & asynchronous way using fs module?

Q18. How to capture command line arguments while executing node js process?

Q19. What is error check first in callback handler defined in Node JS code?

Q20. What are different module pattern in javascript, can you explain common JS modules?

Q21. What is callback hell and how can it be avoided any library which can be used and how to promisify the library?

Q22. What are promises and how to use promises for simple AJAX call or for multiple AJAX calls?

Q23. What is global object in Node JS & how it can used to manage environments in application?

Q24. What are streams and how it's different from normal api response?

Q25. How we can have separate config for development & production environments, configuration file like which manages database connection?

Q26. What are the modules type which node js currently supporting?

Q27. what is the differences between promises, callback & async await?

Q28. What is event loop, is it part of V8 runtime environment and also available on browser?

Q29. what will be the state of Node JS process when event loop is empty and call stack both are empty?

Q30. What is call stack, is it part of V8 runtime environment?

Q31. Have you used yarn as package manager and how its different from NPM?

Q32. What are the tools to deploy node js application on server?

Q33. How to graceful shutdown your Node JS process when something bad happens in code like database connection lost?

Q34. How can you make sure of zero downtime while Node JS deployment of your application?

Q35. What is the use of cluster module and how to use it?

Q36. How can you debug you Node JS application?

Q37. what are streams and why we should use it with large data?

Q38. What are code module in Node JS explain few of them and their use?

Q39. Explain event loop lifecycle and explain few events inside event loop?

Q40. How to prevent Unhandled Exception in Node JS and if they are occurring how to handle them?

Q41. How to convert callback based library to promise based library so instead of writing callback we can write promises?

Q42. What is global object in Node JS how to add object to global variable & how it's different from browser global environment?

Question 1

What is NPM & how its being used with NodeJS?

npm is the package manager for JavaScript based application and the world's largest software registry having thousands of javascript modules.You can Discover packages of reusable code — and assemble them in powerful new ways.

NPM is powerful package manager to install new package for your application. It will give you common js modules.

NPM gives you capability to initialize application using package.json.

NPM and yarn are most popular package manager to manage javascript modules.If you are creating some reusable code and wanted to share with others you can create NPM module and push in to NPM and that module can be used by others.

npm initnpm install -save reactnpm instal -g webpacknpm instal —save-dev gulp

We can install package globally, Locally and install package in dev dependencies.

Global packages will be available system wide and can be accessed on system cli like webpack and webpack-dev-server modules.

npm as package manager install new package for application which can be added globally locally in your application as local module or local dependencies.

Node js is using NPM and its coming with node version as default package manager to manage dependencies locally and globally.

Question 2

What is Node.JS? When should we use Node JS?

Node.js is a server side language based on Google's V8 JavaScript engine.

It is used to build scalable programs and need to run very fast. Its built on top of V8 runtime engine whose baseline in libio & libuv Libraries for C++.

Node js can be used to build api and application required real time interface like reading live data, streaming data and doing socket communication.

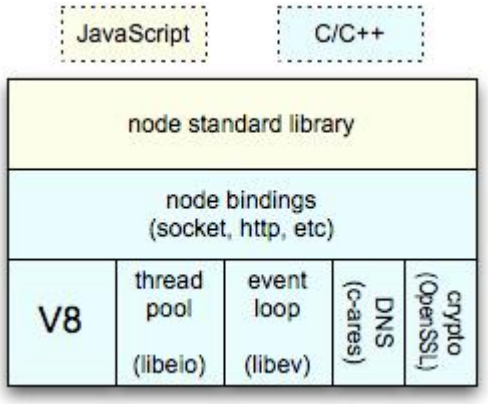
Node js should never be used with CPU intensive tasks like reading huge files and reading lot of data from database and sending it somewhere else.

Node.js is a highly efficient which can scale enough & provide non-blocking I/O running on single thread event loop that was built on top of Google Chrome V8 engine and its ECMAScript.

- Node JS provide scalable applications
- Node JS is server side javascript and single threaded.
- Node JS adds non I/O blocking platform.
- Node JS is built on top of v8 chrome engine.
- Node JS provides faster way to create REST APIs and have good stack of library to support applications.
- Node JS providing faster application development and can be used in microservices like environments.

Question 3

What is of Node js platform stack, what are different libraries it is using?



Node JS using V8 runtime Engine same as chrome is using, it built on top of same chrome engine.

It's an open source JIT(Just In Time) compiler written in c++ which has outperformed PHP, Ruby and python performance wise. V8 compiler compiles Javascript directly into assembly level code. V8 Runtime environment comprises into 3 major component

Compiler : dissects the JS code

Optimizer : Optimizer called crankshaft create abstract syntax tree(AST) which further converts to SSA : static single assignment and gets optimized

Garbage Collector : it removes dead objects from new space and puts into old space. Garbage collector play vital role in keeping Node Js lightweight.

Base library are c++ library and libeio managing thread pool so javascript engine here is single thread but internally its managing thread pool.

1. Libuv/libio : A C++ library

This library handles Node's asynchronous I/O operation and main event loop. There are thread pool reserve in Libuv which handles the thread allocation to individual I/O operations.

On top of c++ library node js has binding with http, socket io binding which are being invoked by core modules of node js like fs, net, dns, socket.io, http.

Node js standard library are written in javascript to access c++ library interface and access interface as Node js will be running on server not on simple browser accessing html and javascript.

Question 4

What is Node JS Architecture?

Node is single threaded and based on non I/O blocking way of dealing with operation.

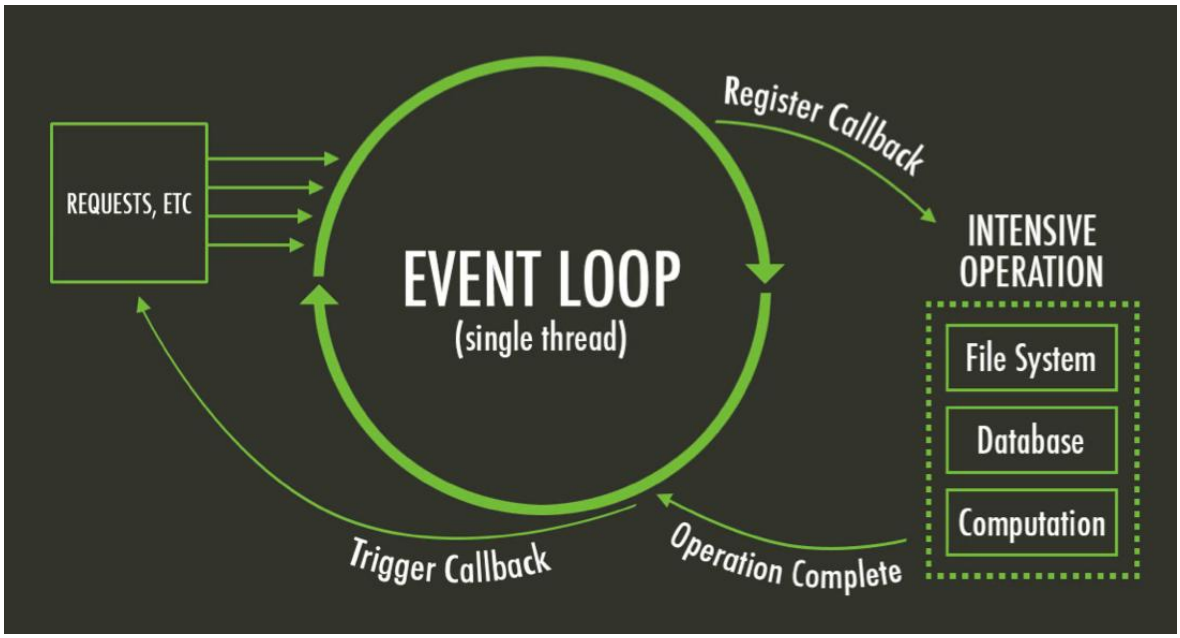
It is fast and scalable while running on single thread and doing I/O operation like database read, file read in asynchronous way using event loop.

Node JS is single threaded or its javascript interface is single threaded, but this is a half truth, actually it is event-driven and single-threaded with background workers. The Event loop is single-threaded but most of the I/O works run on separate threads, because the I/O APIs in Node.js are asynchronous/non-blocking by design, in order to accommodate the event loop.

Node js event loop is feature of Node js base library, Node js is running on single threaded environment and can provide more performance using this single threaded event driven model, Node js manages event loop keeps running in search of asynchronous request.

Node js event loop keeps running and whenever any asynchronous request comes its places in event loop if event loop is not busy and gets processed and further notified once execution over, Once you are getting multiple async requests then it pushed all request to event queue and start processing request one by one without blocking the code execution.

Event loop is a part of Libeio library and running as single thread keeps waiting for async request ina cycle once it sees anything coming it will process it.



Question 5

Explain in depth about event loop mechanism in Node JS?

The Event Loop is a queue of callback functions. When an async function executes like setTimeout, the callback function is pushed to the queue. The JavaScript engine doesn't continue processing the event loop until the code after an async function has executed.

Event-loop is the main part of the node js system. it keeps running and executing as long as node js process are active in memory. It's responsible for handling asynchronous operations in application like http call, I/O operation and database read. These all request will be queued to the event loop waiting to be executed on the next free I/O, on execution completion the event loop will got notified to trigger a callback to the main function.

```
request(`http://www.google.com`, function(error, response, body) {console.log(body);});
console.log('Done!');
```

In above code example, using request module we are making an http call on google.com url. It is asynchronous operation as you will reading data from network. This task will be pushed to event queue if event loop is busy in processing tasks & further once event loop if free, event queue will send that I/O request to event loop for execution. we will get response from the callback added in this code, it will get executed once we have response from I/O task.

Node js runtime execution is not blocked by asynchronous tasks, it will move to the next statement like in above example after running http call using request module it will move to console.log statement. we will see output on console and after sometime we will be notified with this callback with data coming from network request.

Question 6

1. What is the role of package.json and what are NPM scripts?

This file package.json has the information about the project. this gives information to npm that allows it to identify the project as well as handle the project's dependencies either local or devdependencies.

Some of the fields are basic information like name, name, description, author and dependencies, script and some meta tags.

if you install application using npm then all the dependencies listed will be installed as well. Additionally, after installation it create./node_modules directory.

Package.json is just json file having meta information about application. Here main part is npm scripts which are important part and will be executing our application with different commands. Like running node js process either we write node index.js on terminal directly or we just run npm run start.

Which one is better?

We use npm scripts to automate our tasks and list all them together in npm scripts where we can execute them using npm run , npm scripts are powerful you can add pre and post hooks for these tasks. These will work like task runners we used to have like gulp and grunt.

```
{
 "name": "node-js-sample",
 "version": "0.2.0",
 "description": "A sample Node.js app using Express 4",
 "main": "index.js",
 "scripts": {
 "start": "node index.js"
 },
 "dependencies": {
 "express": "⁴.13.3"
 },
 "engines": {
 "node": "4.0.0"
 },
 "repository": {
 "type": "git",
 "url": "https://github.com/heroku/node-js-sample"
 },
```

```
 "keywords": [
 "node",
 "heroku",
 "express"
 ],
 "author": "Mark Pundsack",
 "contributors": [
 "Zeke Sikelianos <zeke@sikelianos.com> (http://zeke.sikelianos.com)"
 ],
 "license": "MIT"
}
```

In this above mentioned package.json we have script tag and having start task, so npm scripts enable execution of this tasks using

npm run start will run node index.js.

Question 7

What is Event-driven programming? And how Node JS is event driven language?

From the name itself its clean that what is event driven something driven by some event so in node js we can write events can trigger something once those event occurs

Event driven programming : Node js is having event loop on single thread and there is always one event will be running and always there will be one handle to handle that event without interruption.

Before getting into code of eventEmitter we should understand why event loop is best example of event driven programming.

Event loop perform two operation in a loop

```
var events = require('events');
var eventEmitter = new events.EventEmitter();
//Create an event handler:
var EventHandler = function () {
 console.log('I hear a voice!');
}
//Assign the event handler to an event:
eventEmitter.on('horror', EventHandler);
//Fire the 'scream' event:
eventEmitter.emit('horror')
```

1. Event detection
2. Event handler triggering

There can be different events like on database read do that or on doing this successfully after that run this code.

Those things can be done using events, event is a core modules where we can emit event from one place and can define handler which can take care of handling that event.

Example below showing events by extending EventEmitter class, its code sample using classes where we have created custom event emitter by extending EventEmitter class and emitting event and capturing it.

```
const EventEmitter = require('events');
class MyEmitter extends EventEmitter {}
```

```
const CustomEmitter = new MyEmitter();
CustomEmitter.on('event', function(a, b) {
 console.log('processed in first iteration',a,b);
});
CustomEmitter.emit('event', 'Hi', 'Hello');
```

Question 8

How Node JS V8 runtime is different from what we have on chrome console?

Chrome console and Node js both are using V8 javascript runtime engine but major difference is Node JS using other c++ core libraries to manage http and socket communication and Chrome V8 engine is mainly browser oriented environments where node js is browserless environment mainly CLI based to execute tasks.

On browser we have access to window, document and console objects and in Node js we don't have document & window objects available, it is server side runtime environment which can be executed from command line. Node js mainly used for creating HTTP server, socket communication or reading or writing real time data.

1. Runtime Environment

Node.js:

- Server-Side: Node.js is designed to run JavaScript on the server side. It enables the creation of scalable network applications.
- Non-Browser Environment: Node.js operates outside of the web browser, allowing it to access the file system, perform networking operations, and interact with the operating system.
- Built-in Modules: Node.js provides a set of core modules (like fs, http, path, etc.) that offer functionalities for file handling, server creation, and more.

Chrome Console:

- Client-Side: The Chrome console is primarily used for debugging and executing JavaScript in the context of a web page.
- Browser Environment: It runs within the browser, thus it has access to the DOM (Document Object Model), CSSOM (CSS Object Model), and various browser-specific APIs like localStorage, sessionStorage, fetch, etc.
- No Direct Access to File System: For security reasons, the browser environment restricts direct access to the user's file system and other OS-level functionalities.

2. APIs and Modules

Node.js:

- Node-Specific APIs: Node.js has APIs for file I/O, networking, and child processes, which are not available in the browser.
- NPM (Node Package Manager): Node.js uses npm to manage packages and dependencies, allowing developers to include third-party libraries and modules.

Chrome Console:

- Web APIs: It can use browser-specific APIs like DOM, fetch, WebSockets, Service Workers, and more.
- No NPM: The console does not support npm or its packages directly. It relies on what is included in the web page or served via web servers.

3. Event Loop and Asynchronous Programming

Node.js:

- Server-Side Event Loop: Node.js uses an event-driven, non-blocking I/O model that is well-suited for real-time applications and handling many concurrent connections.
- Libuv: Node.js uses the libuv library to handle asynchronous operations, providing a consistent API on top of different operating systems.

Chrome Console:

- Client-Side Event Loop: The browser's event loop handles user interactions, rendering, and asynchronous operations (e.g., AJAX requests, timers) within the context of the web page.
- Web Workers: For intensive tasks, browsers use Web Workers to run scripts in background threads.

4. Use Cases

Node.js:

- Server-Side Applications: Ideal for building web servers, APIs, and back-end services.
- CLI Tools: Useful for creating command-line tools and scripts.
- Real-Time Applications: Perfect for chat applications, live data streaming, etc.

Chrome Console:

- Web Development: Used for debugging and testing JavaScript code in the context of web pages.
- Frontend Debugging: Helps developers inspect and manipulate the DOM, monitor network requests, and debug JavaScript errors on the client side.

Question 9

What is the difference between Asynchronous function, Synchronous function or pure functions?

Synchronous function : Those function which do simple execution and don't deal with I/O, These are simple function where we can predict output also and have only basic operation having data manipulation

Asynchronous Function : Special function which deals with network I/O operations like database read, file read or getting data from some api. These operation always takes time while executing and you do not receive instant response from these api.

simple example asynchronous code sample :

```
var userDetails;
function initialize() {
 // Setting URL and headers for request
 var options = {
 url: 'https://api.github.com/users/narenaryan'
 };
 return new Promise(function(resolve, reject) {
 // Do async job
 request.get(options, function(err, resp, body) {
 if (err) {
 reject(err);
 } else {
 resolve(JSON.parse(body));
 }
 })
 })
}// Synchronous code samplefunction foo(){}
function bar(){
 foo();
}
function baz(){
 bar();
}
baz();
```

Question 10

what are different options to write asynchronous code in Node?

Using setTimeout we can run some code after defined time

Using callback : return function from another function after asynchronous task is over.

Using Async module : async module in node js

Using promises : using native promises and wait until promise is resolved

Using some library like bluebird,Q library.

Using async/await : write less line of code by using async await

Node JS Interview Question — Set #01

Question 11

How single threaded Node JS handles concurrency when multiple I/O operations happening?

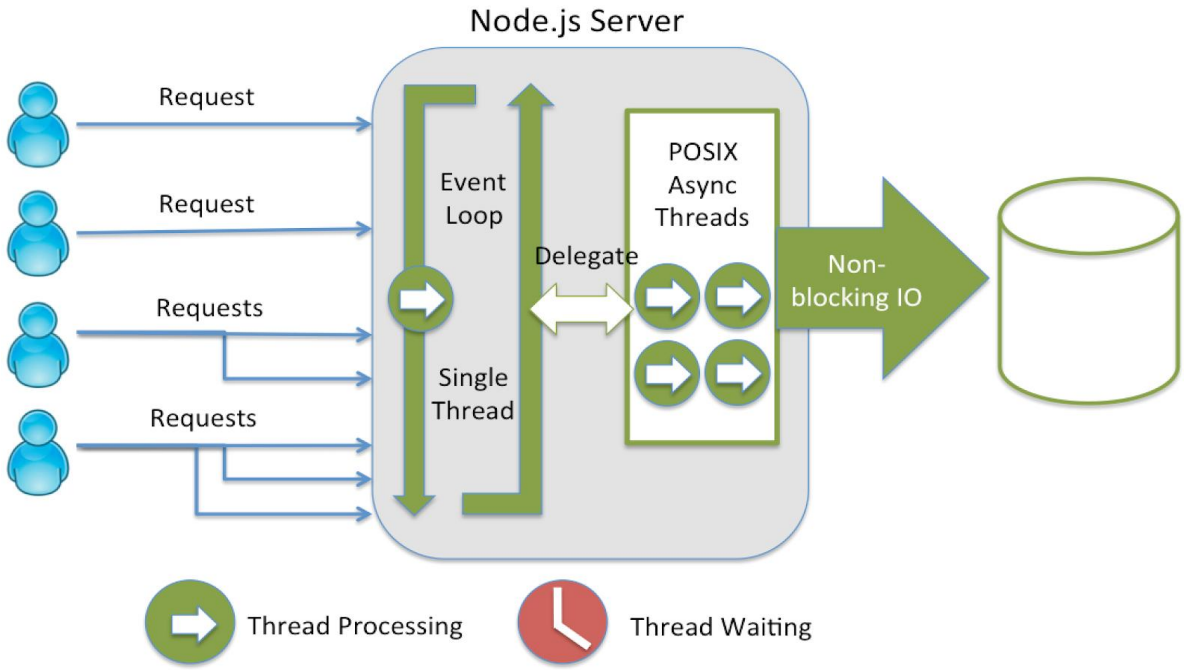
Node provides a single thread to the code we are writing so that code can be written easily and without bottleneck and without I/O blocking Node internally uses multiple POSIX/unix threads for various I/O operations such as network read, database read or file read operations.

When Node apis or code get I/O request it creates a thread from thread pool to perform that asynchronous operation and once the operation is done, it pushes the result to the event queue. On each such event, event loop runs and checks the queue and if the execution stack of Node is empty then it adds the queue result to execution stack.

This is how Node manages concurrency.

Reference : https://strongloop.com/strongblog/node-js-is-faster-than-java/

Request modeling is different, In node js its not creating separate thread for every request, just running event loop and delegating to async thread for running async tasks.



Question 12

What is Node JS callback and how it helps to run Async tasks?

Callback is a Node.js function which can be created with plain javascript code. While writing callback you don't need node js environment. It's just higher order function which takes function as argument and return callback function . This function is used to avoid I/O or network blocking and allows next instruction to run

Node JS handle all asynchronous calls via callback natively. Callbacks are just functions that you pass to other functions. which we call as higher order function in javascript. Example on callback is while reading file using fs node js core module we pass first argument is the path of the file and second argument is a function, which is nothing but a callback function

Node.js use callback function extensively. Node APIs are written to support callbacks of Node JS.

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

Question 13

Explain basic async apis in javascript like setTimeout, setImmediate & setInterval.

The setTimeout & setInterval are the 2 timers functions.these function are being used to create timer function.

setTimeout :- This function is used to delay the execution of code written under it. This will execute once after the defined delay in milliseconds.

After 1 second we will get console message saying "Hello"

```
function sayHi() {
 console.log('Hello');
}
setTimeout(sayHi, 1000);setInterval :-  if you want to execute a function many times or unlimited times then we can use .setInterval() by passing the interval duration.

This function will keep executing after every second and print value of i on console

let i = 0;
function increment() {
i++;
_console_.log(i);
}
var myVar = setInterval(increment, 1000);
// clearInterval(myVar);
```

clearInterval method will be used to clearout the execution of method from setInterval

setImmediate() and setTimeout() are based on the event loop.

Another important method in setImmediate, we use setImmediate if we want to queue the function behind whatever I/O event callbacks that are already in the event queue.

we can use process.nextTick to effectively queue the function at the head of the event queue so that it executes immediately after the current function completes. It queues them immediately after the last I/O handler somewhat like process.nextTick. So it is faster.

Question 14

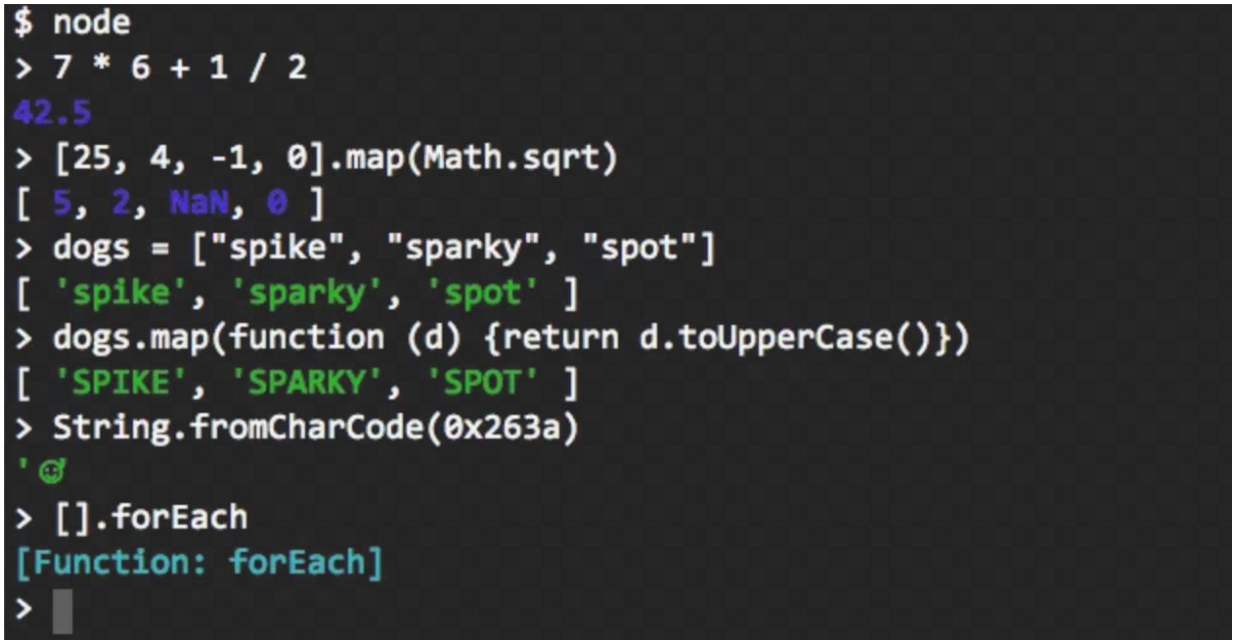
What is REPL in Node JS and how it helps to run code?

Node.js comes with environment called REPL (aka Node shell). REPL stands for Read-Eval-Print-Loop, its easiest way to run node js code on console.

The repl module provides a Read-Eval-Print-Loop (REPL) implementation that is available both as a standalone program or includible in other applications. It can be accessed using:

will give you a result which is similar to the one you will get in the console of Google Chrome browser, it look like chrome console without browser based apis

const repl = require('repl');



Question 15

What are core and important modules in node JS which used frequently and explain difference between core module and user defined module?

Node.js being lightweight framework. All the core modules include minimum functionalities of Node.js. These core modules are compiled into its binary distribution and load automatically when Node.js process starts

> Cluster The cluster module helps us to create children process that runs simultaneously and share the same server port. So using this we can create process running on same port, As node js is single threaded and efficiently use memory by consuming single core only but when we are running on multi core system in such case to take advantage of multi-core systems Cluster module allows you to easily create child processes that each runs on their own single thread, to handle the load.

```
Crypto — To handle OpenSSL cryptographic functions
Dgram — Provides implementation of UDP datagram sockets
Dns — To do DNS lookups and name resolution functions
Domain — Deprecated. To handle unhandled errors
Events — To handle events
Fs — To handle the file system
Http — To make Node.js act as an HTTP server
Https — To make Node.js act as an HTTPS server.
Net — To create servers and clients
Os — Provides information about the operation system
Path — To handle file paths
```

All above mentioned are code module in node js as they are coming bundled with node js installation. User defined module are those which we are creating in node js application and writing module.export in file and again making them require in another file.

Reference : **https://nodejs.org/api/synopsis.html**

Question 16

Explain events in Node JS and how events are helping us to create event driven system.

Every action has reaction similarly every event has it handler to catch the action and take action on it.

In node js we have event emitter which is used to emit event and further that event can be captured to perform some operation.

Events can be compared by simple socker.io example when server broadcast a message it can be captured by all client who are subscribing to that channel.

Node.js has a built-in core module known as "Events", where you can create, fire, and listen for your own events.

Its plain and simple we create events and written handler at another place to capture that event

To include the built-in Events module use the require() method. All event properties and methods are an instance of EventEmitter object available on events so to be able to access these properties and methods we need to create an EventEmitter object:

```
var events = require('events');
var eventEmitter = new events.EventEmitter();
```

Here is the example where we are emitting event name "scream" and there is a handler my eventHandler which is capturing this event and processing task. These events are like fire and forget you fire and forget them.

```
var events = require('events');
var eventEmitter = new events.EventEmitter();//Create an event handler:
var myEventHandler = function () {
 console.log('I hear a scream!');
}//Assign the event handler to an event:
eventEmitter.on('scream', myEventHandler);//Fire the 'scream' event:
eventEmitter.emit('scream');
```

Question 17

How we can read file in synchronous way & asynchronous way using fs module?

The normal behaviour in Node.js is to read in the content of a file in a non-blocking,asynchronous way. That is to tell Node to read the file and return callback once you are done with reading file, there are different events also like file read started or file read over.

We use core module fs and fs.readfile provides asynchronous way of reading file

For this we can use the readFile method of the fs core module

Node js core module fs takes 3 arguments, These arguments are name of the file ('app.txt' in this case), the encoding of the file ('utf8'), and a callback function as argument. This function which is a callback function going to be called when the file-reading operation has finished and we will see file content printed on terminal.

As this operation is non I/O blocking and asynchronous in nature, this will be process using event loop but execution will not be blocked we will see message "file read is over" before getting file contents on terminal

```
var fs = require('fs');
fs.readFile(app.txt, 'utf8', function(err, contents) {
 console.log(contents);
});
console.log('file read is Over');
```

There is another way to use blocking file read operation using **fs.readFileSync** which will read file in synchronous way and it block the execution until unless file read operation is over.

```
var fs = require('fs');
var contents = fs.readFileSync('app.txt', 'utf8');
console.log(contents);console.log('file read is Over');
```

Question 18

How to capture command line arguments while executing node js process?

The arguments are stored in process.argv when you pass args with node command

like node index.js "hello" "world"

```
[runtime] [script_name] [argument-1 argument-2 argument-3 … argument-n]
```

process.argv is 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.

we can get all arguments printed using this Loop

and we can pass information to an application before it starts. This is particularly useful if you want to perform some settings before starting application like passing env and port

```
process.argv.forEach(function (val, index, array) {
 console.log(index + ': ' + val);
});
```

Question 19

What is error check first in callback handler defined in Node JS code?

generally, the first argument to any callback handler is an an error object and there is a reason to pass first argument as error object in callback handler which can be either null or error object so while dealing with callback we can check if we have received null or some error object

if we get error object then we will perform our action based on error. Error handling by a typical callback handler could be as follows:

```
function callback(err, results) {
 // usually we'll check for the error before handling results
 if(err) {
 // handle error somehow and return
 }
 // no error, perform standard callback handling
}
```

This is applied to all callbacks we write in our code and its part of ESLint configuration which force developer to write code in this way.

Question 20

What are different module pattern in javascript, can you explain common JS modules?

In JavaScript, the word "modules" refers to small units of independent, reusable code. They are the foundation of many JavaScript design patterns and are critically necessary when building any non-trivial JavaScript-based application.

We have different module pattern in javascript like commonjs, AMD, UMD and ES6 modules

A CommonJS module is essentially a reusable piece of code can be fetched from either nomjs.com repository or created locally. From module we can exports specific objects, making them available for other modules to require in their programs. While writing node js code you have seen this and may be very familiar with this format.

Using Common JS every JavaScript file stores modules in its own unique module context (just like wrapping it in a closure).We use module.export to export module and require to require that module in another file.

module.export and require syntax we use everywhere in Node js Code, all common js modules are imported in such a way only

```
var app = require('./app')
function myModule() {
 this.hello = function() {
 return 'hello!';
 }
 this.goodbye = function() {
 return 'goodbye!';
 }
}
module.exports = myModule;
```

Question 21

What is callback hell and how can it be avoided any library which can be used and how to promisify the library?

Callback hell refers to a coding style which we use like when we add nested callbacks in application, Lot of nesting of callback functions create callback hell. The code it becomes difficult to debug and understand in such cases we can use other library to overcome with callback hell problem

a. Using promises
b. Yield operator and Generator functions from ES6
c. Modularising code
d. Using async library using async waterfall
e. by not doing nested callback

In javascript most of library supports callback way of writing code like redis-client, mysql-client all such library are callback based so better if we promisify these library and use them with promises

We can use util core module to promisify module in node js like we can promisify fs module which provide callback based operation fileread.

Now callback based file read write operation become promise based and we need to do ".then()" to capture response from resolved promise.

Question 22

What are promises and how to use promises for simple AJAX call or for multiple AJAX calls?

Promises give an alternate way to write asynchronous code and it gives advantages over callback.

We can use promises instead of using callbacks. Promises either return the result of execution or the error/exception. Promises have different state resolved, rejected or pending state. Once promise resolved or rejected .then method gets triggered Promises simply requires the use of <.then()> function which waits for the promise object to return and once you have final state it gets executed. Promise.then() function takes two arguments, both are callback functions first for success callback and another error callbacks.

```
readFileAsync(filePath, {encoding: 'utf8'})
 .then((text) => {
 console.log('CONTENT:', text);
 })
 .catch((err) => {
 console.log('ERROR:', err);
 });
function readFileAsync(){
 return new Promises(function(resolve,reject){
 resolve('some data')
 })
}
```

Question 23

What is global object in Node JS & how it can used to manage environments in application?

The Global keyword represents the global namespace object, we can get what is in global by opening node terminal or you can do console.log(global) in your application.

when we declare variable using let/const those are module specific modules but when we declare without using let/var they gets added to global namespace of application.

We can also add few things on global object like runtime environment configuration

Process, buffer is also part of Global object

Process in very big object having all information about process here we also have global.process.env where we can manage environment specific configuration

```
env:
 { MANPATH: '/Users/tarun.kumar/.nvm/versions/node/v8.0.0/share/man:/usr/local/share/man:/usr/share/man:/Library/Dev
   TERM_PROGRAM: 'Apple_Terminal',
   NVM_CD_FLAGS: '',
   SHELL: '/bin/bash',
   TERM: 'xterm-256color',
   TMPDIR: '/var/folders/ls/32mr9w0x7h96j0vhtrlg_n200000gp/T/',
   Apple_PubSub_Socket_Render: '/private/tmp/com.apple.launchd.Aap3RXcLko/Render',
   TERM_PROGRAM_VERSION: '388.1.1',
   TERM_SESSION_ID: '30843FDC-DDC4-480B-AE6E-D7F331B551CA',
```

In our application we pass these env variable while running node application like

```
Node index.js node_env=local port=5009
```

In this example we will get local when we try to get value of process.env.node_env and will get 5009 when we do process.env.port in our code.

Question 24

What are streams and how it's different from normal api response?

Streams are just flow of data, steam pipes that let you easily read data from a source and pipe it to a destination. so stream are easy to pipe from one source to another, A stream is nothing but an EventEmitter and implements some specials methods. Depending on the applied methods in node js code , a stream becomes Readable, Writable, or Duplex (both readable and writable).

There are different use cases of streams we can use stream to pipe response to api server.

For example we can create file reader stream which read file until data in completed from that file and during that we have some events also like read start or data event or error events

```
var fs = require("fs");
var data = '';
var readerStream = fs.createReadStream('input.txt');
readerStream.setEncoding('UTF8');
readerStream.on('data', function(chunk) {
 data += chunk;
});
readerStream.on('end',function(){
 console.log(data);
});
readerStream.on('error', function(err){
 console.log(err.stack);
});
```

Question 25

How we can have separate config for development & production environments, configuration file like which manages database connection?

There are different option either you can use .dotenv module to manage configuration for application runtime.

We can manage different config file based on different environments like dev.properties qa.properties file

At runtime we should pass process.env.NODE_DNV as either development or production so in code we can load appropriate env file and further we can load its configurations like mongodb url, Mysql connection url which will be different for development and production

we can require that file and can get configuration object and pass them wherever required.

```
var config = {
 production: {
 mongo : {
 url: ''
 }
 },
 dev: {
 mongo : {
 url: ''
 }
 }
}
exports.get = function get(env) {
 return config[env] || config.default;
}
const                              config                              =
require('./config/config.js').get(process.env.NODE_ENV);
const dbconn = mongoose.createConnection(config.mongo.url);
```

Question 26

What are the modules type which node js currently supporting?

In Javascript we have modules like es6, commonjs, AMD,UMD.

The obvious one for Node JS is CommonJS, which is the current module system used by Node JS (the one that uses require and module.exports). CommonJS already is a module system for NodeJS, and ES Modules has to learn to live side by side and interoperate with it

whatever we do today using module.export & require all are common js modules available on npm repository

All module downloaded from npmjs.com or npm repository supports common js style of require and exports

Till now if we want to use ES6 modules in ES6 like using import and export syntax we have to use babel polyfill like babel-register or babel-node as there is no native support for ES6 modules in node js

Node js native doesn't support ES6 modules or code like import/export

But now native support is coming but with slight change, these will be called ESM module with extension of .esm

```
export const spout = 'the spout'
export const handle = 'the handle'
export const tea = 'hot tea'import {handle, spout, tea} from './01-kettle.mjs'console.log(handle) // ==> the handle
console.log(spout) // ==> the spout
console.log(tea) // ==> hot tea
```

Question 27

what is the differences between promises, callback & async await?

Async await has been introduced recently and powerful tool to write asynchronous code in synchronous fashion, async await code look like simple synchronous code and it blocks the event loop and implemented on top of promises only.

Async functions

For async we just need to add it before function name as "async" function

```
async function f() {
 return 1;
}
```

Async before a function always returns a promise. If the code has return in it, then JavaScript automatically wraps it into a resolved promise with that value.

```
let value = await promise;
```

The keyword await makes JavaScript wait until that promise settles and returns its result.

Here's example with a promise that resolves in 1 second:

```
async function f() {
 let promise = new Promise((resolve, reject) => {
 setTimeout(() => resolve("done!"), 1000)
 });
 let result = await promise; // wait till the promise resolves (*)
 alert(result); // "done!"
}
f();
```

Promises are another important tool in javascript to manage asynchronous code.

Promises either return the result of execution or the error/exception. Promises have different state resolved, rejected or pending state. Once promise resolved or rejected .then method gets triggered, Promises simply requires the use of <.then()> function which waits for the promise object to return and once you have final state it gets executed. Promise.then() function takes two arguments, both are callback functions first for success callback and another error callbacks.

Promises allow us to cleanly chain chain subsequent operations while avoiding callback hell and as long as you always return a promise for each of your then blocks, it will continue down the chain.

Promises are now supported in native code so no need to add external library. Promises are just representation of asynchronous code which can be in resolved/rejected state and accordingly .then function will execute with proper callback

Little bit about callback

Node handles all asynchronous operation using callback natively. Callbacks are just functions that you pass to other functions. Example like while reading file using fs core module we pass first argument is the path of the file and second argument is a function, which is nothing but a callback function.

Question 28

What is event loop, is it part of V8 runtime environment and also available on browser?

The event loop is provided by the libuv library. It is not part of V8 runtime env.

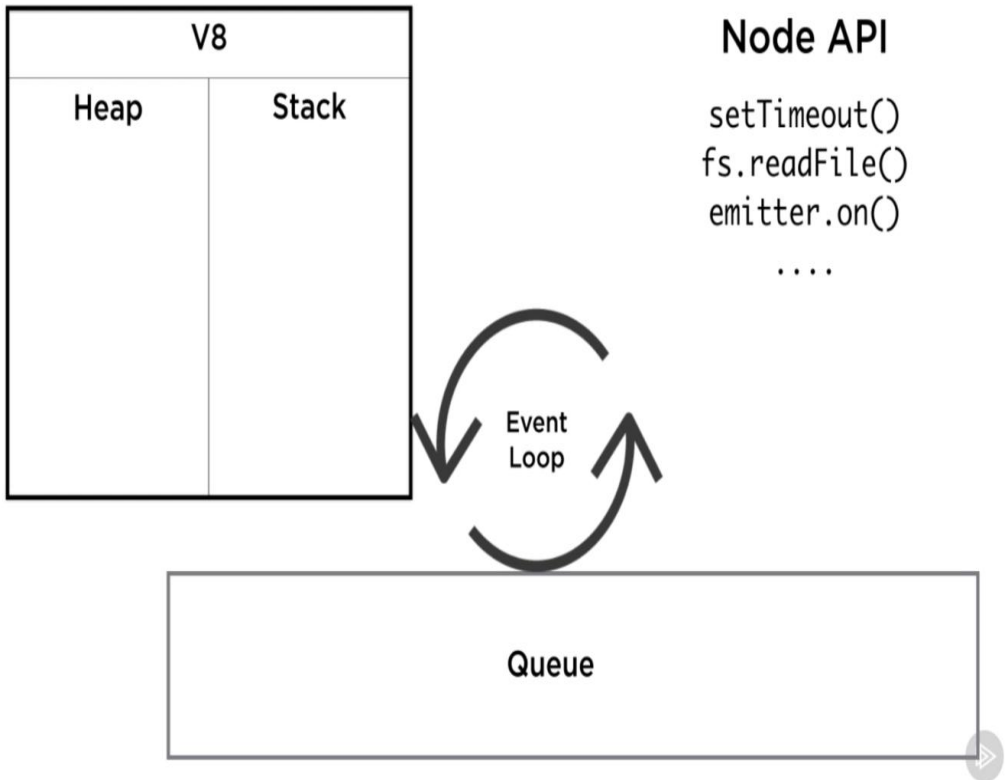
it is single thread entity keeps running in node js process and keep listening the event queue.

The Event Loop is the entity that handles external events and converts them into callback invocations. It is a loop that picks events from the event queues and pushes their callbacks into the Call Stack.

There is only one thread that executes JavaScript code and this is the thread where the event loop is running. The execution of callbacks is done by the event loop.

To understand more on event loop you can use this reference link

https://medium.com/the-node-js-collection/what-you-should-know-to-really-understand-the-node-js-event-loop-and-its-metrics-c4907b19da4c



Question 29

what will be the state of Node JS process when event loop is empty and call stack both are empty?

In such case node js process will exit as it has nothing to process nothing to execute, it is different from any other process, when we start node js process it will start event loop and if there is nothing to execute in event loop it will exit from there.

Node.js Process States

Active State: The Node.js process is actively executing JavaScript code, processing events, and handling asynchronous operations.

Idle State: When there are no pending tasks, timers, or IO operations, the process is idle but still running, waiting for new tasks or events to handle.

Exit State: The Node.js process exits when there are no more pending operations or callbacks to process, effectively terminating the process.

What Happens When Both Event Loop and Call Stack are Empty

Event Loop Empty: The event loop checks for pending events or tasks that need to be processed. If it finds none, it remains idle, waiting for new tasks to be added.

Call Stack Empty: The call stack holds the current function calls to be executed. When it is empty, it indicates that there is no JavaScript code currently being executed.

Possible Outcomes

Process Waits for New Events: If there are no scheduled timers (e.g., setTimeout, setInterval), no pending I/O operations (e.g., file read/write, network requests), and no immediate tasks (e.g., setImmediate), the event loop will wait for new events to process. The Node.js process remains active but idle, consuming minimal resources, and can instantly react when a new event arrives.

Process Exits Gracefully: If the event loop and call stack are both empty and there are no more registered event listeners or pending operations, Node.js will determine that there is nothing left to do and will exit the process gracefully. This happens automatically to conserve system resources.

Example Scenario

Consider the following simple Node.js script:

```
console.log("Hello, World!");

setTimeout(() => {
  console.log("This will never run because the timeout is cleared.");
}, 1000);


// Clearing the timeout to demonstrate an empty event loop.
clearTimeout(timeoutId);

console.log("Script finished execution.");
```

In this script:

"**Hello, World!**" and "**Script finished execution.**" are logged to the console.

The setTimeout is cleared before it can execute, leaving no pending timers.

Once the script completes execution and the event loop finds no pending tasks, Node.js will exit the process gracefully.

Key Points

Idle State: Node.js can remain idle, waiting for new events if the event loop and call stack are empty but there are still potential future events that could occur (e.g., network connections that may receive data).

Exit State: Node.js will exit the process if it determines that there are no more operations to be performed, and no potential future events are expected.

To prevent this situation we always create http server where server keeps telling event loop for listening http request events and event loop is not totally idle.

```
var http=require('http')
var server=http.createServer(serverFn)
server.listen(7000);
```

Question 30

What is call stack, is it part of V8 runtime environment?

Yes call stack is a part of Javascript having either chrome V8 engine or chakra engine.

The JavaScript engine V8 is a single threaded interpreter having a heap and a single call stack. The browser additionally provides some APIs like the DOM, AJAX, and Timers. This is main core functionality which is responsible to execute functions using heap and call stack (Stack which use LIFO pattern)

Call stack operates by principle of Last In, First Out known as LIFO, it says that the last function that gets pushed into the stack is the first to be pop out, when the function returns.It pure stack like data structure based operation.

We can see that in below example where function calling each other.

```
function firstFn(){
 console.log('hello from first Fn');
}
function secondFn(){
 firstFn();
}
function thirdFn(){
 secondFn();
}
thirdFn();
```

1. When secondFn() gets executed, an empty stack frame is created. It is the main entry point of the program.
2. secondFn() then calls firstFn() which is pushed into the stack using LIFO pattern.
3. firstFn() returns and prints "Hello from firstFn" to the console.
4. firstFn() is pop off the stack.
5. The execution order then move to secondFunction().
6. secondFn() returns and print "The end from secondFunction" to the console.
7. secondFn() is pop off the stack, clearing the memory occupied.

Question 31

Have you used yarn as package manager and how its different from NPM?

Yarn is just another package manager for installing and managing javascript libraries for application. Yarn is using the same registry that npm does. That means that every package that is a available on npm is the same on Yarn.

To add a package, run yarn add .

If you need a specific version of the package, you can use yarn add package@version

yarn also have init command

The yarn init command will walk you through the creation of a package.json file to configure some information about your package. This is same as we do npm init using npm package manager

these are following differences we can see

Yarn has a few differences from npm. Yarn is doing caching of all installed packages. Yarn is installing the packages simultaneously and when we install packages with Yarn it look like faster than NPM. Both yarn and NPM downloading packages from same NPM repository.

On the contrary to npm, Yarn provides stability, putting lock on versions of installed packages. The speed is higher while installing packages. It is very important for big projects, which have more dependencies

Question 32

What are the tools to deploy node js application on server?

There are many popular tool to deploy node js app on server which will keep node js application up and running and if there is any issue it will restart process like PM2, forever, supervisord.

PM2 provides :

1. Built in load balancer
2. Multiple instance of application running on same port.
3. Can run application in cluster mode.
4. Can manage deployment of multiple application using single config.
5. Provides multiple deployment options.
6. Provides zero downtime on application deployment.

If you use pm2, you can easily hook it with keymetrics.io monitoring tool to see api statics.

```
npm install -g pm2
pm2 start app.js
```

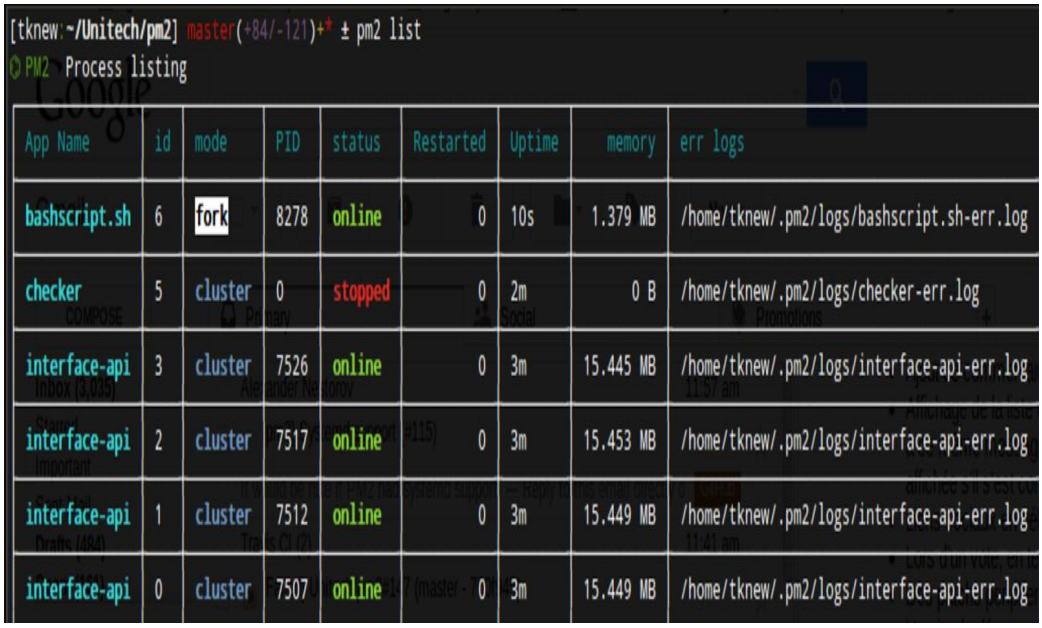
Zero-config Load-Balancer Link

PM2 enable use to create multiple instance to scale up your application by creating instances that share the same server port. Doing this also allow you to restart your app with zero-seconds downtimes.

PM commands to start/stop/delete application instance

```
pm2 start app.js — name “my-api”

pm2 start web.js — name “web-interface”

pm2 stop web-interface. …

pm2 restart web-interface. …

pm2 delete web-interface. …

pm2 restart /http-[1,2]/ …

pm2 list # Or pm2 [list|ls|l|status]
```

pm2 list command showing all available instances and pm2 monit command showing monitoring for all running instances



PM2 has lot of advantages over other tools it gives you everything and its is industry standard for node js deployments.

Question 33

How to graceful shutdown your Node JS process when something bad happens in code like database connection lost?

graceful shutdown means whenever node js process shut down in that case we need to shut process down gracefully by closing all db connection by releasing TCP port and releasing all occupied resources so when node process comes up again there should be no any issues

graceful shot down can be done in different way

Shutdown can happen using some code issue like unhandled promise rejection, some javascript code null check missing or database shutdown or forcefully by user using Ctrl + C

```
process.on('unhandledRejection', ErrorHandler.unhandledRejection);
process.on('SIGINT', ErrorHandler.shutdown);
process.on('uncaughtException', ErrorHandler.onError);process
 .on('unhandledRejection', (reason, p) => {
 console.error(reason, 'Unhandled Rejection at Promise', p);
 // release database connection
 // release resources
 });
```

Question 34

How can you make sure of zero downtime while Node JS deployment of your application?

Zero downtime means whenever we deploy node js application it stop the application and then start it again in this case there is a downtime of some seconds until application gets connected to database like redis, mysql. Zero downtime can't be achieved with single instance when we have huge traffic coming for APIs.

PM2 is powerful tool to manage multiple instances running multiple core of one machine. On multi core system we should always have multiple instance of PM2 to optimally consume multiple cores of system.

1. For having zero downtime we should run application in cluster mode which allows networked Node.js applications (http(s)/tcp/udp server) to be scaled across all CPUs available, without any code modifications. This greatly increases the performance and reliability of your applications, depending on the number of CPUs available.

To enable the cluster mode, just pass the -i option:

pm2 start app.js -i max (max means that PM2 will auto detect the number of available CPUs and run as many processes as possible)

For zero downtime we should use pm2 reload not restart command, reload is different from restart as it will start reloading one by one and not doing reload on all instance together. pm2 restart app-name ,which kills and restarts the process. pm2 reload app-name which restart your workers one by one, and for each worker, wait till the new one has spawned before killing the old one. Using this reload we can serve request by live workers without having any issue in api-services.

This below mentioned configuration "ecosystem.config.js" will create max number of instances based on available core on system and run all instances in cluster mode.

pm2 startOrReload ecosystem.config.js — update-env

This command will start max number of instance if they have not been created yet or reload the existing created instances. — update-env parameter will reload instance with some newly added configuration.

Ecosystem.config.js file:module.exports = {

```
apps: [
{
name: 'api_app',
script: 'app/server.js',
```

```
 instances: “max”
 }
 ]
};
```

Question 35

What is the use of cluster module and how to use it?

cluster module provide a way to create child process. In some cases we may need to have a child process for running some independent process and want to distribute some process. For that purpose we can use Cluster module which will create another child process and that process is created using like forking a process.

The cluster module is core node js module like fs,net module. Cluster module contains a set of functions and properties that help us forking processes to take advantage of multi-core systems. Node js runs on single core system but when we have multi core system and in that case to use that multi core system we should create child process which are equal to number of processor in system

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;
if (cluster.isMaster) {
 masterProcess();
} else {
 childProcess();
}
function masterProcess() {
 console.log(`Master ${process.pid} is running`);
 for (let i = 0; i < numCPUs; i++) {
 console.log(`Forking process number ${i}…`);
 cluster.fork();
 }
 process.exit();
}
```

Question 36

How can you debug you Node JS application?

If you are using vscode like tool for writing node js apps then debugging is easier you can just run debugger in few simple steps.

To start debugging, run your Node.js application with the — inspect flag.

$ node — inspect <your_file>.js

And you need to add chrome dev tools for that so you can launch debugger on chrome on some port and can do debugging like you do for client side javascript code.

Another option is node-inspector

$ npm install -g node-inspector

$ node-debug app.js

where app.js is the name of your main Node application JavaScript file.

The node-debug command will load Node Inspector in your default browser.

Question 37

what are streams and why we should use it with large data?

Streams are collections of items such as we have array as collection. The difference is that streams data might not be available all at same time and not necessary they all fit in existing memory. Stream data are not for synchronous execution. Stream are meant to be received over the time asynchronously.

Stream are powerful tools to send response from apis when you have big data to send.

This makes streams really powerful when working with large amounts of data, or data that's coming from an external source one chunk at a time like file reading of big size. In that case we have to read file in chunks and need to stream that data to send in response.

They also give us the power of sending data in chunks. Just like we use pipe command in linux and send data of one command to another command, we can do exactly the same in Node with streams.

```
const fs = require('fs');
const server = require('http').createServer();server.on('request', (req, res) => {
 fs.readFile('./app.txt', (err, data) => {
 if (err) throw err;

 res.end(data);
 });
});// using streams //
server.on('request', (req, res) => {
 const src = fs.createReadStream('./xpp.txt');
 src.pipe(res);
});
```

Question 38

What are code module in Node JS explain few of them and their use?

'Events',

'fs',

'http',

'https',

'module',

'net',

'os',

'path',

'stream'

child_process

The child_process module provides the ability to spawn child processes in a manner that is similar, but not

identical, to popen(3).

https://nodejs.org/api/child_process.html

cluster

A single instance of Node.js runs in a single thread. To take advantage of multi-core systems the user will

sometimes want to launch a cluster of Node.js processes to handle the load. The cluster module allows you to

easily create child processes that all share server ports.

https://nodejs.org/api/cluster.html

Events

Much of the Node.js core API is built around an idiomatic asynchronous event-driven architecture in which certain

kinds of objects (called "emitters") periodically emit named events that cause Function objects ("listeners") to be Called.

https://nodejs.org/api/events.html

fs

File I/O is provided by simple wrappers around standard POSIX functions. To use this module do require('fs'). All the methods have asynchronous and synchronous forms.

https://nodejs.org/api/fs.html

http

The HTTP interfaces in Node.js are designed to support many features of the protocol which have been traditionally difficult to use. In particular, large, possibly chunk-encoded, messages.

readline

The readline module provides an interface for reading data from a Readable stream (such as process.stdin) one line at a time.

repl

The repl module provides a Read-Eval-Print-Loop (REPL) implementation that is available both as a standalone program or includible in other applications.

Question 39

Explain event loop lifecycle and explain few events inside event loop?

In latest release of V8 event loop is also available in JavaScript engine (V8, spiderMonkey etc). Event loop is part of Libuv library & in reality event-loop is the master which uses the JavaScript engines to execute JavaScript code. event loop runs on separate thread.

When you run node index.js in your console, node start the event-loop and then runs your main module main module from index.js outside the event loop. Once the main module is executed, node will check if the loop is alive if event loop is not alive then node js process simply exit otherwise it will keep listening the event queue

At some point during the event loop, the runtime starts handling the messages on the queue, starting with the oldest one. Once that execution is over event is emitted about completion of task and handed over to handler.

The event loop is a fundamental part of Node.js that allows it to perform non-blocking I/O operations, despite the fact that JavaScript is single-threaded. Understanding the event loop's lifecycle and its different phases helps in grasping how Node.js handles asynchronous operations.

The event loop is a fundamental part of Node.js that allows it to perform non-blocking I/O operations, despite the fact that JavaScript is single-threaded. Understanding the event loop's lifecycle and its different phases helps in grasping how Node.js handles asynchronous operations.

Event Loop Lifecycle

The event loop is responsible for executing code, collecting and processing events, and executing queued sub-tasks. Here's a high-level overview of its lifecycle:

Initialization: When a Node.js application starts, it initializes the event loop, prepares the environment, and begins executing the code.

Execution: The main code is executed in the call stack, and asynchronous operations (like I/O tasks) are offloaded to the event loop.

Event Loop Phases:

The event loop proceeds through multiple phases in each iteration (tick). Each phase has a specific purpose and handles different types of callbacks.

Exit: The event loop continues running as long as there are pending operations or callbacks to handle. When there are no more tasks left, the process exits.

Event Loop Phases

The event loop has six distinct phases, each handling different kinds of tasks:

Timers: This phase executes callbacks scheduled by setTimeout and setInterval.

Pending Callbacks: Executes I/O callbacks deferred to the next loop iteration.

Idle, Prepare: Used internally by Node.js for miscellaneous tasks. This phase is rarely directly interacted with by developers.

Poll: Retrieves new I/O events; executes I/O-related callbacks (except for setImmediate and close callbacks); if the event loop is idle, it will wait for I/O to happen.

Check: Executes setImmediate callbacks. These callbacks are executed after the poll phase has completed.

Close Callbacks: Executes close callbacks, such as when a socket or handle is closed.

Detailed Explanation of Few Events Inside the Event Loop.

1. Timers Phase

Example: setTimeout, setInterval

Behavior: This phase handles the execution of callbacks scheduled by setTimeout and setInterval. If the specified time has elapsed, their respective callbacks are added to the callback queue and executed.

```
setTimeout(() => {

  console.log('Timer phase: setTimeout callback');

}, 1000);
```

2. Pending Callbacks Phase

Example: Callbacks deferred to the next loop iteration by certain I/O operations.

Behavior: Executes callbacks that were deferred to the next iteration of the event loop. These typically include I/O callbacks that couldn't be executed in the poll phase.

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

fs.readFile('somefile.txt', (err, data) => {

  if (err) throw err;

  console.log('Pending callbacks phase: File read callback');

});
```

3. Poll Phase

Example: I/O operations

Behavior: This is the phase where the event loop retrieves new I/O events and executes their callbacks. It will also handle I/O-related tasks until the queue is exhausted or a specified timeout is reached.

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

const server = net.createServer((socket) => {

  socket.end('Handled by poll phase\n');

});

server.listen(3000, () => {

  console.log('Poll phase: Server listening on port 3000');

});
```

4. Check Phase

Example: setImmediate

Behavior: Executes callbacks scheduled by setImmediate. These callbacks are invoked after the poll phase, ensuring they run immediately after I/O events are processed.

```
setImmediate(() => {

  console.log('Check phase: setImmediate callback');
```

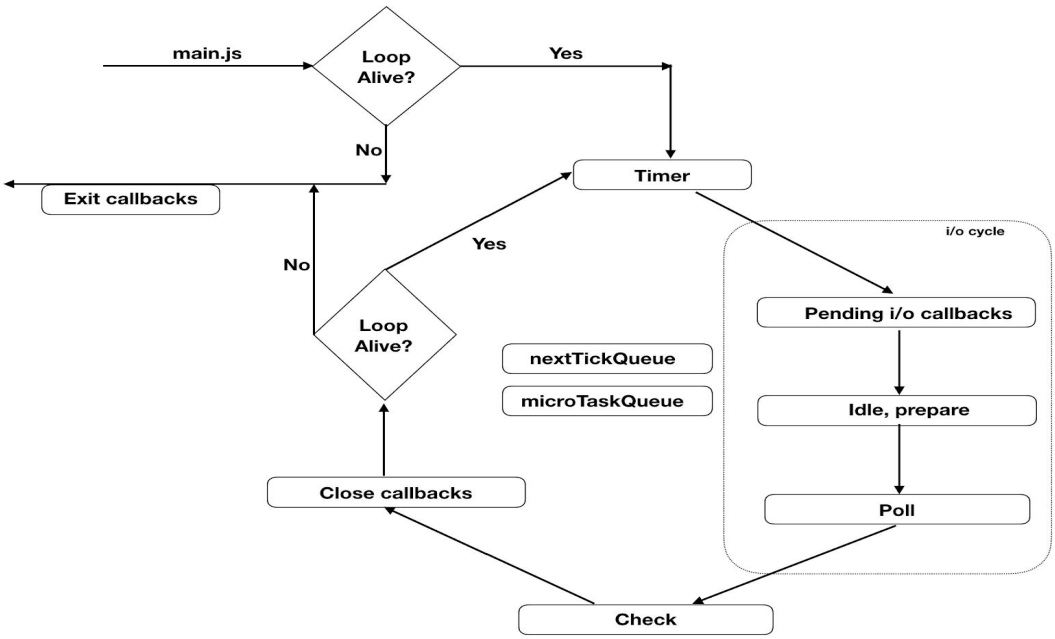
```
});
```

5. Close Callbacks Phase

Example: socket.on('close', ...)

Behavior: Handles the execution of callbacks related to closing resources like sockets and file descriptors.

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

const socket = new net.Socket();

socket.on('close', () => {

  console.log('Close callbacks phase: Socket closed');

});


socket.destroy();  // This will trigger the 'close' event
```



Question 40

How to prevent Unhandled Exception in Node JS and if they are occurring how to handle them?

Node.js event loop runs on a single thread and uncaught exceptions are critical issue and need to be aware of when developing applications.

Silently Handling Exceptions

Most of the people let node.js server(s) silently swallow up the errors.

Silently handling the exception

```
process.on('uncaughtException', function (err) {
 console.log(err);
});
```

This is bad, it will work but:

In this case Root cause of problem will remains unknown, as such will not contribute to resolution of what caused the Exception (Error).

In case of node application having database connection (pool) which gets closed for some reason will result in constant propagation of errors, meaning that server will be running but it will not reconnect to db. So you should write a code which can manage and do not generate Unhandled exception and for debugging purpose you can caught those process and identify the cause.

Question 41

How to convert callback based library to promise based library so instead of writing callback we can write promises?

This is an important aspect and this is being used for many library like redis-client, mysql-client, we can use bluebird.promisifyAll to convert any callback based library to promise based.

```
db.notification.email.find({subject: 'promisify callback'}, (error, result) => {
 if (error) {
 console.log(error);
 }
 // normal code here
});
```

It is using bluebird's promisifyAll method to promisify what is conventionally callback-based library.

After applying these change promise-based methods names wil have Async appended to them:

```
let email = bluebird.promisifyAll(db.notification.email);
email.findAsync({subject: 'promisify callback'}).then(result => {
 // normal code here
})
.catch(console.error);
```

Same thing can be done for redis library which is callback based library

```
const redis = require('redis');
bluebird.promisifyAll(redis);
client.getAsync('data:key').then(function(res) {
 console.log(res); // => 'bar'
});
```

Question 43

What is global object in Node JS how to add object to global variable & how it's different from browser global environment?

In browsers when we do console.log(this) it represent the window object and in node JS the global scope of a module is the module itself, so when you define a variable in the global scope of your node JS module, it will be local to this module.

In node js global represents global scope only you can add variable in global object it will be available on node js process and can be accessed in any other local module.

We can add some common configuration to global object like mysql connection object or logger object as we will using these objects on different places in application.

```
const mysql = require('mysql2');
global.connection = null;
try {
 global.connection = mysql.createConnection(global.configuration.db);
} catch (err) {
 throw Error(err);
}
global.connection.connect((err) => {
 if (err) throw err;
});
global.connection.on('error', (err) => {
 logger.info(`Cannot establish a connection with the database (${err.code})`);
});
module.exports = global.connection;
```