**Q1)** **What is Node Js?**

**A)**. 1. Node.js is a platform for the JavaScript language that is centered around asynchronous network programming. It contains a set of libraries to help you develop server-side applications with JavaScript

2. Under the hood, Node.js is running on V8, a JavaScript engine developed by Google. **(OR)**

Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

**Q2)** **Why Node Js?**

**A)** Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.

**Q3) What is Event driven?**

**A)** Event-driven means that the server only reacts when an event occurs. This allow us to create high performance, highly scalable, “real-time” applications.

**Q4)** **Blocking I/O** (VS) **NON-Blocking I/O**

**A) Blocking I/O:**

In the blocking method, user2's data request is not initiated until user1's data is printed to the screen.

If this was a web server, we would have to start a new thread for every new user. But JavaScript is single-threaded (not really, but it has a single-threaded event loop, which we’ll discuss a bit later). So this would make JavaScript not very well suited for multi-threaded tasks.

That’s where the non-blocking part comes in.

**Non-blocking I/O:**

On the other hand using a non-blocking request, you can initiate a data request for user2 without waiting for the response the request for user1. You can initiate both requests in parallel.

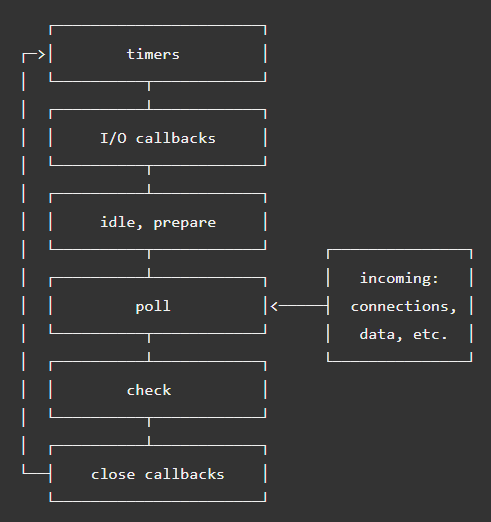
This non-blocking I/O eliminates the need for multi-threading, since the server can handle multiple requests at the same time.

**Q5)** **What is Event loop in Node.js work? And How does it work?**

A)The Event loop handles all async callbacks. Node.js (or JavaScript) is a single-threaded, event-driven language. This means that we can attach listeners to events, and when a said event fires, the listener executes the callback we provided.

Whenever we are call setTimeout, http.get and fs.readFile, Node.js runs this operations and further conitnue to run other code without waiting for the output. When the operation is finished, it receives the output and runs our callback function.

So all the callback functions are queued in an loop, and will run one-by-one when the response has been received.



There are six phases in the Event loop but one works internally. Below is an overview of each phase from the Node.js doc.

**timers**: this phase executes callbacks scheduled by setTimeout() and setInterval().

**I/O callbacks**: executes almost all callbacks with the exception of close callbacks, the ones scheduled by timers, and setImmediate().

**idle, prepare**: only used internally.

**poll**: retrieve new I/O events; node will block here when appropriate.

**check**: setImmediate() callbacks are invoked here.

**close callbacks**: such as socket.on(‘close’).

**Q6)**  **Node.js Is Single Threaded : What’s The Fuss About It ?**

A) Yes! you heard it right, Node is single threaded and it is doing magical things with this model. Some of the popular server side technology like PHP, ASP.NET, Ruby & Java Servers all follow Multi-threaded where each client request results in the instantiation of a new thread or even a process, but Node. js, requests are run on the same thread with even shared resources. All Node JS applications uses **“Single Threaded Event Loop Model”** architecture to handle multiple concurrent clients.

So Yes NodeJS is single threaded, but this is a half truth, actually it is event-driven and single-threaded with background workers. The main 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.

**Architecture:**

**Single Threaded Event Loop Model Processing Steps:**

1. Clients Send request to Web Server.

2. Node JS Web Server internally maintains a Limited Thread pool to provide services to the Client Requests.

3. Node JS Web Server receives those requests and places them into a Queue. It is known as “Event Queue”.

4. Node JS Web Server internally has a Component, known as “Event Loop”. Why it got this name is that it uses indefinite loop to receive requests and process them. (See some Java Pseudo code to understand this below).

5. Event Loop uses Single Thread only. It is main heart of Node JS Platform Processing Model.

6. Even Loop checks any Client Request is placed in Event Queue. If no, then wait for incoming requests for indefinitely.

7. If yes, then pick up one Client Request from Event Queue

8. Starts process that Client Request

9. If that Client Request Does Not requires any Blocking IO Operations, then process everything, prepare response and send it back to client.

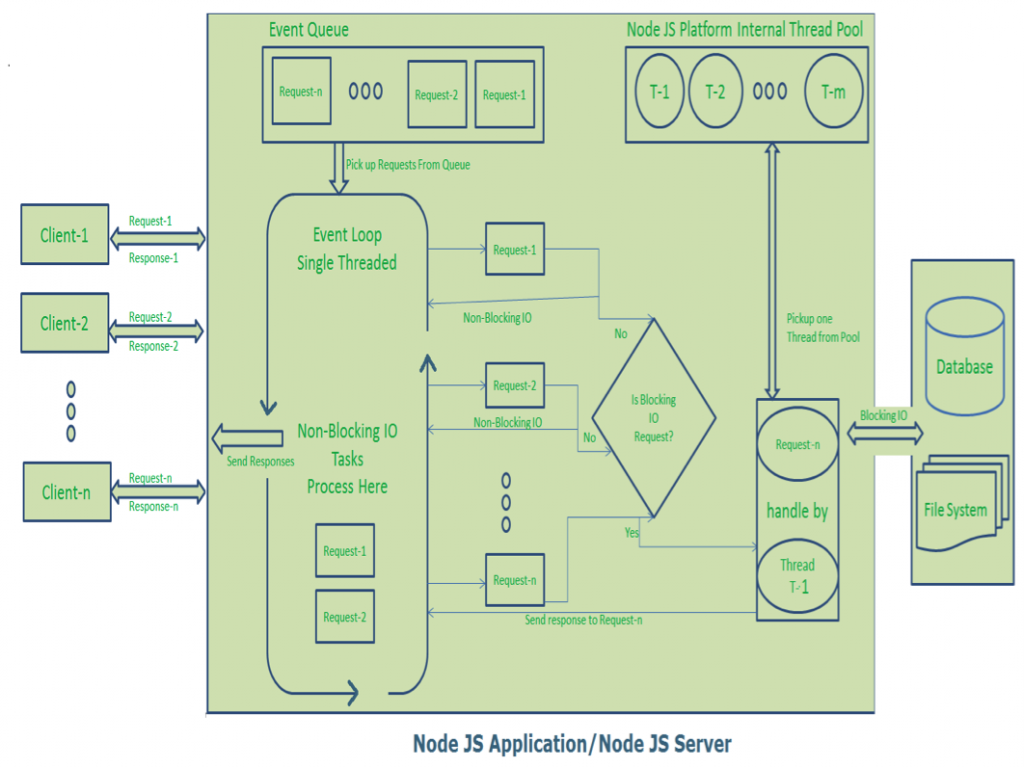
10. If that Client Request requires some Blocking IO Operations like interacting with Database, File System, External Services then it will follow different approach.

11. Checks Threads availability from Internal Thread Pool.

12. Picks up one Thread and assign this Client Request to that thread.

13. That Thread is responsible for taking that request, process it, perform Blocking I/O operations, prepare response and send it back to the Event Loop.

14. Event Loop in turn, sends that Response to the respective Client.



**Node JS Architecture – Single Threaded Event Loop Advantages:**

1. Handling more and more concurrent client’s request is very easy.

2. Even though our Node JS Application receives more and more Concurrent client requests, there is no need of creating more and more threads, because of Event loop.

3. Node JS application uses less Threads so that it can utilize only less resources or memory

-----------------------------------------------------\*\*\*\*\*\*\*\*\*\*\*\*\*\*---------------------------------------------------

**Node.js Console - REPL**

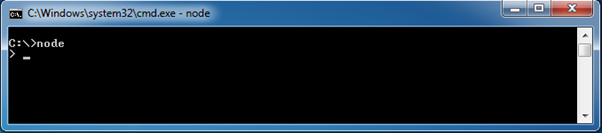
Node.js comes with virtual environment called REPL (aka Node shell). REPL stands for Read-Eval-Print-Loop. It is a quick and easy way to test simple Node.js/JavaScript code

**A read function**, which accepts an expression from the user and parses it into a data structure in memory .

**An eval function**, which takes the data structure and evaluates.

**A print function**, which prints the result.

**A loop function**, which runs the above three commands until termination



> 10 + 20   
30

**REPL Command** **Description**

.help -> Display help on all the commands

tab Keys -> Display the list of all commands.

Up/Down Keys -> See previous commands applied in REPL.

.save filename -> Save current Node REPL session to a file.

.load filename -> Load the specified file in the current Node REPL session.

ctrl + c -> Terminate the current command.

ctrl + c (twice) -> Exit from the REPL.

ctrl + d -> Exit from the REPL.

.break -> Exit from multiline expression.

.clear -> Exit from multiline expression.

**CURD OPERATIONS:**

1. **Creating Server:**

var express=require("express");

var app=express();

var http=require('http');

var server=http.createServer(app);

port=process.env.port || 3002;

server.listen(port,function(){

console.log("server started on port : " + port );

});

2. **body-parser** : body-parser extract the entire body portion of an incoming request stream and exposes it on **req.body.**

npm install body-parser –save

This module provides the following parsers:

* [JSON body parser](https://github.com/expressjs/body-parser#bodyparserjsonoptions)
* [Raw body parser](https://github.com/expressjs/body-parser#bodyparserrawoptions)
* [Text body parser](https://github.com/expressjs/body-parser#bodyparsertextoptions)
* [URL-encoded form body parser](https://github.com/expressjs/body-parser#bodyparserurlencodedoptions)

bodyParser = require('body-parser');

// support parsing of application/json type post data

app.use(bodyParser.json());

//support parsing of application/x-www-form-urlencoded post data

app.use(bodyParser.urlencoded({ extended: false }));

bodyParser.urlencoded([options])

Returns middleware that only parses urlencoded bodies and only looks at requests where the Content-Type header matches the type option. This parser accepts only UTF-8 encoding of the body and supports automatic inflation of gzip and deflate encodings. A new body object containing the parsed data is populated on the request object after the middleware (i.e. req.body). This object will contain key-value pairs, where the value can be a string or array (when extended is false), or any type (when extended is true).

**3.** [**Cookie-parser()**](https://github.com/expressjs/cookie-parser) : a middleware that parse cookie header and populate req.cookies

Parse Cookie header and populate req.cookies with an object keyed by the cookie names. Optionally you may enable signed cookie support by passing a secret string, which assigns req.secret so it may be used by other middleware.

**Installation**

npm install cookie-parser

The following is an example for setting and reading cookies using the [cookie-parser](https://github.com/expressjs/cookie-parser) module:

var express = require('express');

var cookieParser = require('cookie-parser'); // module for parsing cookies

var app = express();

app.use(cookieParser());

app.get('/setcookie', function(req, res){

// setting cookies

res.cookie('username', 'john doe', { maxAge: 900000, httpOnly: true });

return res.send('Cookie has been set');

});

app.get('/getcookie', function(req, res) {

var username = req.cookies['username'];

if (username) {

return res.send(username);

}

return res.send('No cookie found');

});

app.listen(3000);

// Cookies that have not been signed

console.log('Cookies: ', req.cookies)

// Cookies that have been signed

console.log('Signed Cookies: ', req.signedCookies)

**4. MongoDB connection:**

npm install mongoose

// mongodb connection

var mongoose=require('mongoose');

var mongoDB = 'mongodb://localhost:27017/wisdomjobs'; //Set up default mongoose connection

mongoose.connect(mongoDB,{ useNewUrlParser: true }).then(() => console.log('Mongo DB connected Success fully.....'))

.catch(err => console.log(err));

**5. Creating Schema**

var mongoose=require("mongoose");

const Schema=mongoose.Schema;

const UsersSchema=new Schema({

name:String, //(or) name:{type:String,required:true}

email:String,

mobile:Number,

gender:Number,

})

mongoose.model("users",UsersSchema)

**6. Creating Routes:**

**Path: Router/user.js'**

var express=require('express');

var mongoose=require("mongoose");

var router=express.Router();

var usermodel=require('../models/User');

router.get('/', function(req, res) {

res.send('User inserted successfully');

});

module.exports = router;

**app.js:**

var userscreate=require('./Router/user');

app.use('/user', userscreate);

**7.insert data into database using mongodb**

// insert data into database

router.post("/create", (req, res) => {

//console.log(req.body);

mongoose.model('users').create({

name: req.body.name,

email: req.body.email,

mobile: req.body.mobile,

gender: req.body.gender

}, function (err, user) {

console.log(user);

if (err) {

res.send("not inserted");

} else {

var json = {

"status": 200,

"message": "user created successfully"

}

res.send(json);

}

})

});

**“users”** is a table name from ‘mongoose.model("users",UsersSchema) ‘

<http://localhost:3002/user/create> // postman url , method: POST

**in the body-raw send like this:**

{"name":"kumarswamy","email":"kumar2@gmail.com","mobile":123456789,"gender":"male"}

{"name":"kumarswamy","email":"kumar2@gmail.com","mobile":123456789,"gender":"male"}

**8.Getting data from database:**

router.get("/userslist", (req, res) => {

mongoose.model('users').find({}, function (err, user) {

if (err) {

res.send("problem with getting data ");

}

else {

var jsonlist = {

"status": "200",

"message": "data displayed successfully",

"data": user

}

res.send(jsonlist);

}

})

})

<http://localhost:3002/user/userslist> // postman url

**result:**

{

"status": "200",

"message": "data displayed successfully",

"data": [

{

"\_id": "5b8e4653c2790c0dfc74ff12",

"name": "kumarswamy",

"email": "kumar1@gmail.com",

"mobile": 123456789,

"gender": "male",

"\_\_v": 0

},

{

"\_id": "5b8e467ff63f8309d0819e58",

"name": "kumarswamy",

"email": "kumar2@gmail.com",

"mobile": 123456789,

"gender": "male",

"\_\_v": 0

}

]

}

**9. Update Data from database :**

// update data from database

router.put('/:id', (req,res,next)=>{

const id = req.params.id;

mongoose.model('users').findByIdAndUpdate(id, req.body, function(err,userupdate) {

console.log(err);

if (err) {

res.send("no data found belongs to id");

}

else {

var jsonupdate = {

"status": "200",

"message": "data updated successfully with id:" + id,

"data": userupdate

}

res.send(jsonupdate);

}

});

});

<http://localhost:3002/user/5b8e467ff63f8309d0819e58>

{"name":"kumarswamy","email":"kumar2222@gmail.com","mobile":123456789,"gender":"male"}

**10. Delete data from database by using id**

// delete data from database

router.delete('/:id', (req,res,next)=>{

const id = req.params.id;

mongoose.model('users').findByIdAndRemove(id, req.body, function(err,userdelete) {

console.log(err);

if (err) {

res.send("no data found belongs to id for delete");

}

else {

var jsondel = {

"status": "200",

"message": "data deleted successfully with id:" + id,

"data": userdelete

}

res.send(jsondel);

}

});

});

<http://localhost:3002/user/5b8e467ff63f8309d0819e58>

**To delete all records:**

/ delete all rercords

router.delete('/delete/all', (req, res) => {

mongoose.model('users').remove({}, function (err, userdelete) {

console.log(err);

if (err) {

res.send("no data found belongs to deleteAll");

}

else {

var jsondel = {

"status": "200",

"message": "All Records deleted successfully:",

"data": userdelete

}

res.send(jsondel);

}

});

});

<http://localhost:3002/user/delete/all>

**11. To get the count:**

mongoose.model('users').count({}, function(err, count) {

console.log('Count is ' + count);

});

**12. Find the data exist in database or not:**

mongoose.model('users').findOne({ email: req.body.email }, function (err, user) { // for checking user exist!

if (user) {

res.send("User already Exist");

}

**13. Server Side Validations:**

**App.js**

*npm install express-validator;*

var validator = require('express-validator');

app.use(validator());

**user.js**

req.checkBody("name", "Enter a name.").notEmpty();

req.checkBody("name", "Enter minimum length.").isLength({ min: 5, max:10 });

req.checkBody("name", "Enter alphabates only .").matches(/^[a-zA-Z]{5,10}$/, "i"); // allows only alphabates of length 5 and maximum 10

req.checkBody("email", "Enter email address.").notEmpty();

req.checkBody("email", "Enter a valid email address.").isEmail();

req.checkBody("mobile", "Enter a mobile number.").notEmpty();

req.checkBody("mobile", "Enter a mobile number only numbers.").matches(/^[0-9]{10}$/, "i"); //allows only numbers and 10 letters only

req.checkBody("gender", "Enter a gender.").notEmpty();

var errors = req.validationErrors();

if (errors) {

res.send(errors);

return;

} else {

// insertion code here

}

**14. Callback functions:**

**A)** 1.A callback function, also known as a higher-order function, is a function that is passed to another function (let’s call this other function “otherFunction”) as a parameter, and the callback function is called (or executed) inside the otherFunction. A callback function is essentially a pattern (an established solution to a common problem),and therefore, the use of a callback function is also known as a callback pattern.`

2. When a function simply accepts another function as an argument, this contained function is known as a callback function.

Callback functions can be named or anonymous functions

**1. Named callback functions**

function greeting(name) {

console.log(`Hello ${name}, welcome to Scotch!`);

}

function introduction(firstName, lastName, callback) {

const fullName = `${firstName} ${lastName}`;

callback(fullName);

}

introduction('Chris','Nwamba', greeting); // introduction is called the callback function

output:// Hello Chris Nwamba, welcome to Scotch!

**2. Anonymous callback function**

  setInterval(function() {

   console.log('hello!');

  }, 1000);

**Note**: The callback function is not run unless called by its containing function, it is called back. Hence, the term **call back function.**

**Callback hell:**

**Multiple functions can be created independently and used as callback functions. These create multi-level functions. When this function tree created becomes too large, the code becomes incomprehensible sometimes and is not easily refactored. This is known as callback hell. Let’s see an example:**

// a bunch of functions are defined up here

// lets use our functions in callback hell

function setInfo(name) {

address(myAddress) {

officeAddress(myOfficeAddress) {

telephoneNumber(myTelephoneNumber) {

nextOfKin(myNextOfKin) {

console.log('done'); //let's begin to close each function!

};

};

};

};

}

* **Promises:**

**A promise is used to handle the asynchronous result of an operation. JavaScript is designed to not wait for an asynchronous block of code to completely execute before other synchronous parts of the code can run. For instance, when making API requests to servers, we have no idea if these servers are offline or online, or how long it takes to process the server request.**

**With Promises, we can defer execution of a code block until an asynchronous request is completed. This way, other operations can keep running without interruption.**

**Promises have three states:**

**Pending: This is the initial state of the Promise before an operation begins**

**Fulfilled: This means the specified operation was completed**

**Rejected: The operation did not complete; an error value is usually thrown**

**Creating a Promise:**

**The Promise object is created using the new keyword and contains the promise; this is an executor function which has a resolve and a reject callback. As the names imply, each of these callbacks returns a value with the reject callback returning an error object.**

const promise = new Promise(function(resolve, reject) {

// promise description

})

**Example:**

const weather = true

const date = new Promise(function(resolve, reject) {

if (weather) {

const dateDetails = {

name: 'Cubana Restaurant',

location: '55th Street',

table: 5

};

resolve(dateDetails)

} else {

reject(new Error('Bad weather, so no Date'))

}

});

If weather is true, resolve the promise returning the data dateDetails, else return an error object with data Bad weather, so no Date.

**Using Promises:**

Using a promise that has been created is relatively straightforward; we chain .then() and .catch() to our Promise like so:

date

.then(function(done) {

// the content from the resolve() is here

})

.catch(function(error) {

// the info from the reject() is here

});

.**then()** receives a function with an argument which is the resolve value of our promise.

**.catch** returns the reject value of our promise.

**Note:** Promises are asynchronous. Promises in functions are placed in a micro-task queue and run when other synchronous operations complete.

* **Node.js Event Emitter:**

1. Node.js allows us to create and handle custom events easily by using events module. Event module includes **Event Emitter** class which can be used to raise and handle custom events.

2. The event emitter class has two methods.

1. On

2. emit

The following example demonstrates Event Emitter class for raising and handling a custom event.

Example: Raise and Handle Node.js events:

npm install event-emitter

// get the reference of EventEmitter class of events module

var events = require('events');

//create an object of EventEmitter class by using above reference

var em = new events.EventEmitter();

//Subscribe for FirstEvent

em.on('FirstEvent', function (data) {

console.log('First subscriber: ' + data);

});

// Raising FirstEvent

em.emit('FirstEvent', 'This is my first Node.js event emitter example.');

In the above example, we first import the 'events' module and then create an object of EventEmitter class. We then specify event handler function using on() function. The on() method requires name of the event to handle and callback function which is called when an event is raised.

The emit() function raises the specified event. First parameter is name of the event as a string and then arguments. An event can be emitted with zero or more arguments. You can specify any name for a custom event in the emit() function.

You can also use addListener() methods to subscribe for an event as shown below.

var emitter = require('events').EventEmitter;

var em = new emitter();

//Subscribe FirstEvent

em.addListener('FirstEvent', function (data) {

console.log('First subscriber: ' + data);

});

//Subscribe SecondEvent

em.on('SecondEvent', function (data) {

console.log('First subscriber: ' + data);

});

// Raising FirstEvent

em.emit('FirstEvent', 'This is my first Node.js event emitter example.');

// Raising SecondEvent

em.emit('SecondEvent', 'This is my second Node.js event emitter example.');

* **Streams And Buffers:**

**What is Stream?**

Streams are objects that lets you read data from a source or write data to the destination in a continuous fashion.

**Problems with Large Data**:

Speed: Too slow because it has to load all the requests.

Buffer Limit : 1 GB

**Stream Benefits:**

1. Abstraction for continuous chunking of data.
2. No need to wait for the entire resource to load.

**Stream is Used In:**

1. HTTP request & responses

2. Standard input/output(stdin & stdout)

3. File reads and write

**Types of Streams:**

There are four fundamental stream types within Node.js:

**Writable** - streams to which data can be written (for example, fs.createWriteStream()).

**Readable** - streams from which data can be read (for example, fs.createReadStream()).

**Duplex** - streams that are both Readable and Writable (for example, net.Socket).

**Transform** - Duplex streams that can modify or transform the data as it is written and read

(for example, zlib.createDeflate()).

**Readable Stream :**

Readable Stream is used for read operations.

Standard input streams has data going into the applications.This is achieved through the read operation. Input typically comes from the keyboard used to start the process.

const fs = require('fs');

let data = '';

// Create a readable stream

let readableStream = fs.createReadStream('input.txt');

// Set the encoding to be utf8.

readerStream.setEncoding('UTF8');

// Handle stream events --> data, end,

readableStream.on('data', function(chunk) {

data += chunk;

});

readableStream.on('end', function(){

console.log(data);

});

**Writable Stream:**

Writable Stream is used for write operations.

Standard output streams contain data going out of the applications.To write to stdout, we use the write function.

process.stdout.write('A Simple Message \n');

**Duplex Stream :**

This is a Stream which can be used for both read and write operations.

**Transfer Stream:**

A type of duplex stream where the output is computed based on input.

**What is Piping in a Stream?**

Piping is a process in which we provide the output of one stream as the input to another stream. It is normally used to get data from one stream and to pass the output of that stream to another stream. There is no limit on piping operations.

const fs = require('fs');

// Create a readable stream

let readableStream = fs.createReadStream('input.txt');

// Create a writable stream

let writeableStream = fs.createWriteStream('output.txt');

// Pipe the read and write operations

// read input.txt and write data to output.txt

readerStream.pipe(writerStream);

console.log('End of the Process');

**Buffer in Node.js:**

Node.js provides Buffer class which provides instances to store raw data.

**We can create a Buffer in the following way…**

// Create an empty buffer of size 10.

// A buffer that only can accommodate 10 bytes.

const buf1 = Buffer.alloc(10);

// Create a buffer with content

const buf2 = Buffer.from("hello buffer");

//create an uninitiated Buffer of 10 octets

let bufferOne = new Buffer(10);

//create a Buffer from a given array

let bufferTwo = new Buffer([10, 20, 30, 40, 50]);

//create a Buffer from a given string

let bufferThree = new Buffer('Simply Easy Learning');

**Working on Buffer:**

let buffer = Buffer.alloc(26);

for(let i=0; i<26; i++){

buffer[i]=i+97;

}

console.log(buffer.toString('utf8')); // a, b, c.....z

const buf = new Buffer(10); // size is 10

console.log(buf); // Prints: <Buffer 00 00 00 00 00 00 00 00 00 00>

**ex:**

const buf = Buffer.from('hello world', 'ascii');

console.log(buf.toString('hex')); // Prints: 68656c6c6f20776f726c64

console.log(buf.toString('base64')); // Prints: aGVsbG8gd29ybGQ=

Example: Create a single Buffer from a list of three Buffer instances

const buf1 = Buffer.alloc(10);

const buf2 = Buffer.alloc(14);

const buf3 = Buffer.alloc(18);

const totalLength = buf1.length + buf2.length + buf3.length;

console.log(totalLength); // Prints: 42

const bufA = Buffer.concat([buf1, buf2, buf3], totalLength);

console.log(bufA); // Prints: <Buffer 00 00 00 00 ...>

console.log(bufA.length); // Prints: 42

**Important Urls:**

[**https://medium.com/@vigowebs/frequently-asked-node-js-interview-questions-and-answers-b74fa1f20678**](https://medium.com/@vigowebs/frequently-asked-node-js-interview-questions-and-answers-b74fa1f20678)

[**https://www.onlineinterviewquestions.com/node-js-interview-questions/#.W40Tf84zbIU**](https://www.onlineinterviewquestions.com/node-js-interview-questions/#.W40Tf84zbIU)

[**https://www.journaldev.com/7462/node-js-architecture-single-threaded-event-loop**](https://www.journaldev.com/7462/node-js-architecture-single-threaded-event-loop)

[**https://scotch.io/courses/10-need-to-know-javascript-concepts/callbacks-promises-and-async**](https://scotch.io/courses/10-need-to-know-javascript-concepts/callbacks-promises-and-async)

[**https://stackoverflow.com/questions/3707746/cannot-understand-node-js?noredirect=1&lq=1**](https://stackoverflow.com/questions/3707746/cannot-understand-node-js?noredirect=1&lq=1)

[**https://medium.com/tensult/stream-and-buffer-concepts-in-node-js-87d565e151a0**](https://medium.com/tensult/stream-and-buffer-concepts-in-node-js-87d565e151a0)

[**https://www.youtube.com/watch?v=ZEaP\_PGWiDw**](https://www.youtube.com/watch?v=ZEaP_PGWiDw) **(File uploading)**