1. What is node

Node.js is a platform built on Google Chrome's JavaScript V8 engine for easily building fast and 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.

1. Why the Name of node is node

Node actually means a point (of intersection) in a network or diagram at which lines or pathways intersect. Node is a single-threaded, single-process system which enforces shared-nothing design with OS process boundaries. It has rather good libraries for networking. I believe this to be a basis for designing very large distributed programs. The “nodes” need to be organized: given a communication protocol, told how to connect to each other. In the next couple months we are working on libraries for Node that allow these networks.

1. Difference between node and java

|  |  |
| --- | --- |
| Java | Node |
| Synchronous | Async |
| Uses multithreading | Single Threaded |
| Process based | Event based |
| CPU intensive applications can be made with java | CPU intensive applications **cannot** be made |
| slow | node is faster than java |

1. How node is single threaded (why to use node , why single threaded)

Node works on single thread mechanism it achieves the fast performance with single thread with the use of eventloop and callbacks

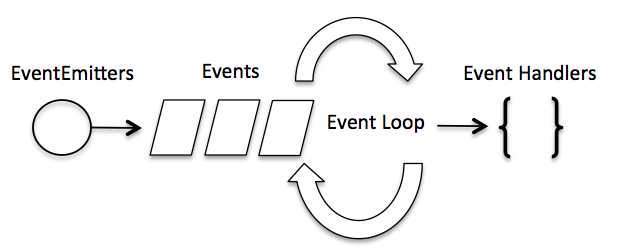
and event loop keeps on eye for any incoming request

as soon as new request is coming it throw the particular event

and keep it in queue for processing.

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

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



1. How node is fast with single thread

Because of event loop and call-backs

As soon as Node starts its server, it simply initiates its variables, declares functions and then simply waits for event to occur.

1. How node works(Threads concept)

Node JS Platform does not follow Request/Response Multi-Threaded Stateless Model. It follows Single Threaded with Event Loop Model. As Node JS follows this architecture, it can handle more and more concurrent client requests very easily.

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

* Clients Send request to Web Server.
* Node JS Web Server internally maintains a Limited Thread pool to provide services to the Client Requests.
* Node JS Web Server receives those requests and places them into a Queue. It is known as “Event Queue”.
* 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).
* Event Loop uses Single Thread only. It is main heart of Node JS Platform Processing Model.
* Even Loop checks any Client Request is placed in Event Queue. If no, then wait for incoming requests for indefinitely.
* If yes, then pick up one Client Request from Event Queue
  1. Starts process that Client Request
  2. If that Client Request Does Not requires any Blocking IO Operations, then process everything, prepare response and send it back to client.
  3. If that Client Request requires some Blocking IO Operations like interacting with Database, File System, External Services then it will follow different approach
     + Checks Threads availability from Internal Thread Pool
     + Picks up one Thread and assign this Client Request to that thread.
     + That Thread is responsible for taking that request, process it, perform Blocking IO operations, prepare response and send it back to the Event Loop
     + Event Loop in turn, sends that Response to the respective Client.

6. What is event loop

An event loop is an internal loop in that processes system and user events

To process and handle external events and to convert them into callback invocations an event loop is used. So, at I/O calls, node.js can switch from one request to another.

Node js is a single threaded application but it support concurrency via concept of event and callbacks. As every API of Node js are asynchronous and being a single thread, it uses async function calls to maintain the concurrency. Node uses observer pattern. Node thread keeps an event loop and whenever any task get completed, it fires the corresponding event which signals the event listener function to get executed.

7. What is loopback

**LoopBack** is an open source Node.**js** framework built on top of Express

It is faster to do project using Loopback than express

It generates API code automatically(folder structure), makes design and testing easier and so that developers can focus more on user experience and business logic. It simplifies connecting to Oracle/MongoDB/SQL Server/MySQL legacy and new data.

8. What is callback == http://callbackhell.com/

Callback is an asynchronous equivalent for a function. A callback function is called at the completion of a given task. Node makes heavy use of callbacks. All APIs of Node are written is such a way that they supports callbacks. For example, a function to read a file may start reading file and return the control to execution environment immediately so that next instruction can be executed. Once file I/O is complete, it will call the callback function while passing the callback function, the content of the file as parameter. So there is no blocking or wait for File I/O. This makes Node.js highly scalable, as it can process high number of request without waiting for any function to return result.

9. What is callback hell

When there are so many Callback functions inside callback that becomes callback hell. This is not a good coding as code becomes unreachable due to this.

10. How call back hell is removed from node

* Don't nest functions. Give them names and place them at the top level of your program
* Handle **every single error** in every one of your callbacks. Use a linter like [standard](http://standardjs.com/) to help you with this.
* **Modularization:** Create reusable functions and place them in a module to reduce the cognitive load required to understand your code. It’s like splitting your code into small pieces.

11. What is promises

Promises are used to avoid callback hell

A Promise object represents a value that may not be available yet, but will be resolved at some point in the future. It allows you to write asynchronous code in a more synchronous fashion. For example, if you use the promise API to make an asynchronous call to a remote web service you will create a Promise object which represents the data that will be returned by the web service in future. The caveat being that the actual data is not available yet. It will become available when the request completes and a response comes back from the web service. In the meantime the Promise object acts like a proxy to the actual data. Furthermore, you can attach callbacks to the Promise object which will be called once the actual data is available.

If everything is successful, the **promise** is fulfilled by calling resolve() . In case of an error, reject() is called with an Error object. This indicates that the **promise** is rejected.

## **The API**

To get started, let’s examine the following code which creates a new Promise object.

if (window.Promise) { // Check if the browser supports Promises

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

//asynchronous code goes here

});

}

We start by instantiating a new Promise object and passing it a callback function. The callback takes two arguments,resolve and reject, which are both functions. All your asynchronous code goes inside that callback. If everything is successful, the promise is fulfilled by calling resolve(). In case of an error, reject() is called with an Error object. This indicates that the promise is rejected.

Now let’s build something simple which shows how promises are used. The following code makes an asynchronous request to a web service that returns a random joke in JSON format. Let’s examine how promises are used here.

if (window.Promise) {

console.log('Promise found');

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

var request = new XMLHttpRequest();

request.open('GET', 'http://api.icndb.com/jokes/random');

request.onload = function() {

if (request.status == 200) {

resolve(request.response); // we got data here, so resolve the Promise

} else {

reject(Error(request.statusText)); // status is not 200 OK, so reject

}

};

request.onerror = function() {

reject(Error('Error fetching data.')); // error occurred, reject the Promise

};

request.send(); //send the request

});

console.log('Asynchronous request made.');

promise.then(function(data) {

console.log('Got data! Promise fulfilled.');

document.getElementsByTagName('body')[0].textContent = JSON.parse(data).value.joke;

}, function(error) {

console.log('Promise rejected.');

console.log(error.message);

});

} else {

console.log('Promise not available');

}

Another Example

function dieToss() {

return Math.floor(Math.random() \* 6) + 1;

}

console.log('1');

var promise = new RSVP.Promise(function(fulfill, reject) {

var n = dieToss();

if (n === 6) {

fulfill(n);

} else {

reject(n);

}

console.log('2');

});

promise.then(function(toss) {

console.log('Yay, threw a ' + toss + '.');

}, function(toss) {

console.log('Oh, noes, threw a ' + toss + '.');

});

console.log('3');

This snippet prints output similar to the following:

1

2

3

Oh, noes, threw a 4.

Or, if we get lucky, we see:

1

2

3

Yay, threw a 6.

In the previous code, the Promise constructor callback contains the asynchronous code used to get data the from remote service. Here, we just create an Ajax request to <http://api.icndb.com/jokes/random> which returns a random joke. When a JSON response is received from the remote server, it is passed to resolve(). In case of any error, reject() is called with an Error object.

When we instantiate a Promise object we get a proxy to the data that will be available in future. In our case we are expecting some data to be returned from the remote service at some point in future. So, how do we know when the data becomes available? This is where the Promise.then() function is used. This function takes two arguments: a success callback and a failure callback. These callbacks are called when the Promise is settled (i.e. either fulfilled or rejected). If the promise was fulfilled, the success callback will be fired with the actual data you passed to resolve(). If the promise was rejected, the failure callback will be called. Whatever you passed to reject() will be passed as an argument to this callback.

Try this [Plunkr](http://plnkr.co/edit/ilf9xtDqrimWxZd77yLI?p=preview) example. Simply refresh the page to view a new random joke. Also, open up your browser console so that you can see the order in which the different parts of the code are executed. Note that a promise can have three states:

* pending (not fulfilled or rejected)
* fulfilled
* rejected

The Promise.status property, which is code-inaccessible and private, gives information about these states. Once a promise is rejected or fulfilled, this status gets permanently associated with it. This means a promise can succeed or fail only once. If the promise has already been fulfilled and later you attach a then() to it with two callbacks, the success callback will be correctly called. So, in the world of promises, we are not interested in knowing when the promise is settled. We are only concerned with the final outcome of the promise.

## **Chaining Promises**

It is sometimes desirable to chain promises together. For instance, you might have multiple asynchronous operations to be performed. When one operation gives you data, you will start doing some other operation on that piece of data and so on. Promises can be chained together as demonstrated in the following example.

function getPromise(url) {

// return a Promise here

// send an async request to the url as a part of promise

// after getting the result, resolve the promise with it

}

var promise = getPromise('some url here');

promise.then(function(result) {

//we have our result here

return getPromise(result); //return a promise here again

}).then(function(result) {

//handle the final result

});

The tricky part is that when you return a simple value inside then(), the next then() is called with that return value. But if you return a promise inside then(), the next then() waits on it and gets called when that promise is settled.

## **Handling Errors**

You already know the then() function takes two callbacks as arguments. The second one will be called if the promise was rejected. But, we also have a catch() function which can be used to handle promise rejection. Have a look at the following code:

promise.then(function(result) {

console.log('Got data!', result);

}).catch(function(error) {

console.log('Error occurred!', error);

});

This is equivalent to:

promise.then(function(result) {

console.log('Got data!', result);

}).then(undefined, function(error) {

console.log('Error occurred!', error);

});

Note that if the promise was rejected and then() does not have a failure callback, the control will move forward to the next then() with a failure callback or the next catch(). Apart from explicit promise rejection, catch() is also called when any exception is thrown from the Promise constructor callback. So, you can also use catch() for logging purposes. Note that we could use try...catch to handle errors, but that is not necessary with promises as any asynchronous or synchronous error is always caught by catch().

As the [Promise.prototype.then()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then) and [Promise.prototype.catch()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/catch) methods return promises, they can be chained—an operation called composition.

12. What is singleton?

A Singleton is an object which can only be instantiated one time. Repeated calls to its constructor return the same instance and in this way one can ensure that they don't accidentally create, say, two Users in a single User application

The **Singleton** Pattern limits the number of instances of a particular object to just one. **This single instance is called the singleton**. **Singletons** are useful in situations where system-wide actions need to be coordinated from a single central place. An example is a database connection pool

13. What is opensource

Its **source** code is available with a license (It’s generally free) in which the copyright holder provides the rights to study, change, and distribute the **software** to anyone and for any purpose. Meaning that other developers can see how it works and add to it. Examples of open source products include Open Office, the internet browser [Mozilla Firefox](http://en.wikipedia.org/wiki/Mozilla_Firefox), Wikipedia, the [GNU/Linux](http://en.wikipedia.org/wiki/Linux) operating system and its derivative [Android](http://en.wikipedia.org/wiki/Android_(operating_system)), an operating system for mobile devices .the idea behind open source software is that users are effectively co-developers, suggesting ways to improve it and helping to hunt out bugs and problems. This means that if you wish, you can modify it to your own needs, port it to new operating systems and share it with others.

14. What is an event?

Event gets triggered when a particular action is done.

15. What is event driven programming, event emitter class?

Event-driven programming is application flow control that is determined by events or changes in state. The general implementation is to have a central mechanism that listens for events and calls a callback function once an event has been detected

In event driven model 3 rules:

1) Event emitter : like cell phonetower

2) Event consumer : apply reaction as soon as event is present

3) Event channels(Streams) :channel means storms which the event is transmitted from emitter to consumer...i.e Streams

For instance: a [net.Server](https://nodejs.org/api/net.html" \l "net_class_net_server) object emits an event each time a peer connects to it; a [fs.ReadStream](https://nodejs.org/api/fs.html" \l "fs_class_fs_readstream) emits an event when the file is opened; a [stream](https://nodejs.org/api/stream.html) emits an event whenever data is available to be read.

To add events to Object use : "events.js" core modules from node.js

var eventEmitter = require("events");

All objects that emit events are instances of the EventEmitter class.

The eventEmitter.on() method is used to register listeners, while the eventEmitter.emit() method is used to trigger the event.

EventEmitter provides multiple properties like **on** and **emit**. **on** property is used to bind a function with the event and **emit** is used to fire an event or  trigger the event.

const EventEmitter = require('events');

class MyEmitter extends EventEmitter {}

const myEmitter = new MyEmitter();

myEmitter.on('event', function(){

console.log('an event occurred!');

});

myEmitter.emit('event');

The EventListener calls all listeners synchronously in the order in which they were registered by .on method This is important to ensure the proper sequencing of events and to avoid race conditions or logic errors. When appropriate, listener functions can switch to an asynchronous mode of operation using the setImmediate() or process.nextTick() methods:

const EventEmitter = require('events');

class MyEmitter extends EventEmitter {};

const myEmitter = new MyEmitter();

myEmitter.on('event',function (a, b) {

setImmediate(function(){

console.log('this happens asynchronously');

});

});

myEmitter.emit('event', 'a', 'b');

## Handling events only once

Using the eventEmitter.once() method, it is possible to register a listener that is called at most once for a particular event. Once the event is emitted, the **listener is unregistered and then called**.

const myEmitter = new MyEmitter();

var m = 0;

myEmitter.once('event', function(){

console.log(++m);

});

myEmitter.emit('event');

// Prints: 1

myEmitter.emit('event');

// **Ignored**

## **Error events**[**#**](https://nodejs.org/api/events.html#events_error_events)

When an error occurs within an EventEmitter instance, the typical action is for an 'error' event to be emitted. These are treated as special cases within Node.js.

If an EventEmitter does not have at least one listener registered for the 'error' event, and an 'error' event is emitted, the error is thrown, a stack trace is printed, and the Node.js process exits.

const myEmitter = new MyEmitter();

myEmitter.emit('error', new Error('whoops!'));

// Throws and crashes Node.js

To guard against crashing the Node.js process, a listener can be registered on the [process object's uncaughtException event](https://nodejs.org/api/process.html#process_event_uncaughtexception) or the[domain](https://nodejs.org/api/domain.html) module can be used. (Note, however, that thedomainmodule has been deprecated)

const myEmitter = new MyEmitter();

process.on('uncaughtException', (err) => {

console.log('whoops! there was an error');

});

myEmitter.emit('error', new Error('whoops!'));

// Prints: whoops! there was an error

As a best practice, listeners should always be added for the 'error' events.

const myEmitter = new MyEmitter();

myEmitter.on('error', (err) => {

console.log('whoops! there was an error');

});

myEmitter.emit('error', new Error('whoops!'));

// Prints: whoops! there was an error

The EventEmitter class is defined and exposed by the events module:

const EventEmitter = require('events');

All EventEmitters emit the event 'newListener' when new listeners are added and 'removeListener' when existing listeners are removed. The EventEmitter instance will emit its own 'newListener' event **before** a listener is added to its internal array of listeners. The 'removeListener' event is emitted **after**the listener is removed.

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

<http://www.baloo.io/blog/2013/11/30/node-event-driven-programming/>

http://www.tutorialspoint.com/nodejs/nodejs\_event\_emitter.htm

19. What is the use of libuv library and why it is used?

libuv is a multi-platform support library with a focus on asynchronous I/O. In node.js “non-blocking” means that its IO is non-blocking. Node uses “libuv” to handle its IO in a platform-agnostic way. Node.js encapsulates libuv to handle asynchronous events.

( libuv is a library used to handle asynchronous events )

20. Why to use sudo in npm install

If you get an EACCES error, you should [fix your permissions](https://docs.npmjs.com/getting-started/fixing-npm-permissions). You could also try using **sudo**, but this **should be avoided**:

sudo npm install -g jshint

(If we don’t have permissions then use sudo, but using sudo is not recommended)

22. Why async.waterfall and how event loop works with async function.

# async.waterfall is used to do things in series(one after another)

function transformFile(inPath, outPath, done) {

async.waterfall([

function(callback) {

fs.readFile(file1, 'utf8', callback); }

function(data, callback) {

service.transform(data, callback); },

function(transformed, callback) {

fs.writeFile(transformed, callback); }

], done);

}

#### Promises

Thanks to .then's behavior when returning a promise, we can chain async operations without any helper tools:

function transformFile(input, output) {

return fs.readFileAsync(input, 'utf8')

.then(service.transformAsync)

.then(fs.writeFileAsync);

}

transformFile(fileIn, fileOut).done(function() {

console.log("All ok!");

}, function(err) {

console.error(err);

});

The resulting promise is fulfilled when all the promises in the chain are fulfilled or is rejected with an error when the first error is encountered.

23. What is asynchronous

It means not in sync, not running at same time. (non blocking)

24. What is javascript and why to use it.

JS is a single threaded programming lang, its object-oriented

Its fast, Being client-side, JavaScript is very fast because any code functions can be run immediately instead of having to contact the server and wait for an answer

**25. What is closure in JavaScript**

A **closure** is an **inner function** that has access to the outer (enclosing) function's variables—scope chain. The **closure** has three scope chains: it has access to its own scope (variables defined between its curly brackets), it has access to the outer function's variables, and it has access to the global variables.

Example

function showName (firstName, lastName) {

​var nameIntro = "Your name is ";

// this inner function has access to the outer function's variables, including the parameter​

​ function makeFullName () {

​ return nameIntro + firstName + " " + lastName;

}

return makeFullName ();

}

showName ("Michael", "Jackson"); // Your name is Michael Jackson

34.What is shrinkwrap

**NPM** itself has a feature called "**shrinkwrap**" that. locks down the versions of a package's dependencies so that you can control exactly which versions of each dependency will be used when your package is installed.

When you run npm shrinkwrap in a project after run­ning npm install, it cre­ates a file called npm-shrinkwrap.json which lists the exact pack­age ver­sions of all the installed pack­ages in the entire hier­ar­chy.

35. Types of http method

|  |  |
| --- | --- |
| 1 | **GET**  The GET method is used **to retrieve information** from the given server using a given URI. Requests using GET should only retrieve data and should have no other effect on the data. |
| 2 | **HEAD**  Same as GET, but transfers the status line and header section only. |  |
| 3 | **POST**  A POST request is used **to send** data to the server, for example, customer information, file upload, etc. using HTML forms. |  |
| 4 | **PUT**  **Replaces all** current representations of the target resource with the uploaded content. |  |
| 5 | **DELETE**  **Removes all** current representations of the target resource given by a URI. |  |
| 6 | **CONNECT**  Establishes a tunnel to the server identified by a given URI |  |
| 7 | **OPTIONS**  Describes the communication options for the target resource. |  |
| 8 | **TRACE**  Performs a message loop-back test along the path to the target resource. |  |

36. What is option method in http?

Used by client to find out the methods supported by the web server

1. Types of Async in JavaScript

Async is a utility module used for asynchronous programming.

http://caolan.github.io/async/

https://www.airpair.com/javascript/async-javascript-libraries

**Async.series**

Run the functions in the tasks collection in series, each one running once the previous function has completed. If any functions in the series pass an error to its callback, no more functions are run, and main callback is immediately called with the value of the error. Otherwise, main callback receives an array of results when tasks have completed.

var async = require('async');

async.series({

one:function(callback) {

console.log('Starting 5 second task');

setTimeout( function() {

console.log('Just finshed 5 seconds');

callback(null, 55);

}, 5000);

} ,

two:function(callback) {

console.log('Starting 2 second task');

setTimeout( function() {

console.log('Just finshed 2 seconds');

callback(null, 22);

}, 2000);

}

}, //series function ends

function(err, results) { //main callback

console.log('After all '+results.one);

// results is now equal to: {one: 55, two: 55}

});

Output is

Starting 5 second task

Just finshed 5 seconds

Starting 2 second task

Just finshed 2 seconds

After all 55

**Async.waterfall**

Runs the tasks array of functions in series, each passing their results to the next in the array. However, if any of the tasks pass an error to their own callback, the next function is not executed, and the maincallback is immediately called with the error

Difference between waterfall and series is – in series no need to pass output of first to second only the second function is executed after 1st, even if there is a timeout difference.

**Async.** **parallel**

Run the tasks collection of functions in parallel, without waiting until the previous function has completed. If any of the functions pass an error to its callback, the main callback is immediately called with the value of the error. Once the tasks have completed, the results are passed to the finalcallback as an array.

parallel is about kicking-off I/O tasks in parallel, not about parallel execution of code. If your tasks do not use any timers or perform any I/O, they will actually be executed in series. Any synchronous setup sections for each task will happen one after the other.

If we run the program given in series example with Async.parellel then output will be

That means the function gets executed when its requirements satisfies , it doesn’t wait for previous to complete.

Starting 5 second task

Starting 2 second task

Just finshed 2 seconds

Just finshed 5 seconds

After all 55

**Async.** **auto**

Determines the best order for running the functions in tasks, based on their requirements. Each function can optionally depend on other functions being completed first, and each function is run as soon as its requirements are satisfied.

If any of the functions pass an error to their callback, the auto sequence will stop

If we run the program given in series example with Async.auto then output will be

Starting 5 second task

Starting 2 second task

Just finshed 2 seconds

Just finshed 5 seconds

After all 55

1. What is async.auto

In Async.auto functions execute in manner they are written and value will be stored in the corresponding variable assigned to statements.

For eg:

async.auto({

initialTask: function(callback) {

//Do some operations

callback(null, name, initialModels);

},

task1: ['initialTask', function(callback, results) {

var models = results.initialTask[1];

//Add some more data to models

callback(null, models);

}],

task2: ['initialTask', function(callback, results) {

var models = results.initialTask[1];

//Add some more data to models

callback(null, models);

}],

task3: ['initialTask', function(callback, results) {

var models = results.initialTask[1];

//Add some more data to models

callback(null, models);

}],

finalTask: ['task1', 'task2', 'task3', function(callback, results) {

//Here the followings are the same: results.initialTask[1], results.task1[0], results.task2[0], results.task3[0]

}]

});

Basically async.auto will execute all the functions in the order it deems necessary. So in this case initialTask will be called first. Then task1, task2, and task3 will be called in parallel. Finally finalTask will be called with the results. The reason all the values are the same is related to JavaScript's [call-by-sharing](http://en.wikipedia.org/wiki/Evaluation_strategy#Call_by_sharing), meaning if you change a function parameter itself, then it won't affect the item that was fed into the parameter. If you change the internals of the parameter, it will carry up to the item.

39. why we use body-parser

body-parser extracts the entire body portion of an incoming request stream and exposes it on req.body

$ npm install body-parser

API:

var **express =** require('express')

var bodyParser **=** require('body-parser')

var app **=** express()

*// parse application/json*

app.use(bodyParser.**json**())

app.use(function (req, res) {

  res.setHeader('Content-Type', 'text/plain')

  res.write('you posted:\n')

var username=req.body.name;

  res.end(JSON.stringify(**req.body**, null, 2))

)}

40. Why to use multer

Multer is a node.js middleware for handling multipart/form-data.

It is written on top of [busboy](https://github.com/mscdex/busboy) for maximum efficiency.

**NOTE**: Multer will not process any form which is not multipart (multipart/form-data).

npm install multer

var express **=** require('express')

var multer  **=** require('multer')

var upload **=** multer({ dest**:** 'uploads/' })

var app **=** express()

app.post('/profile', upload.single('avatar'), function (req, res, next) {

*// req.file is the `avatar` file*

*// req.body will hold the text fields, if there were any*

})

41. Difference between Function and Procedure of SQL

Difference. Function must return a value but in Stored Procedure it is optional( Procedure can return zero or n values). Functions can have only input parameters for it whereas Procedures can have input/output parameters .Functions can be called from Procedure whereas Procedures cannot be called from Function.

what is procedures in **mysql?**

42. What is triggers

A **trigger** is a special kind of stored procedure that automatically executes when an event occurs in the database server. DML **triggers** execute when a user tries to modify data through a data manipulation language (DML) event. DML events are INSERT, UPDATE, or DELETE statements on a table or view.

43. how to get the second highest salary from database

JavaScript

var arr=[3,8,9,1,2];

var maxN=Math.max.apply(null,arr);

console.log('first max '+maxN);

arr.splice(arr.indexOf(maxN));

maxN=Math.max.apply(null,arr);

console.log('second max '+maxN);

OR

mysql

Select max from Emp where sal< (select max from emp)

OR second least

**package** com.td.examples;

**public** **class** Second\_least\_sala {

**public** **static** **void** main(String[] args) {

//**first sort the array**

**int** a[] = { 10, 9, 3, 2, 5 };

**int** t = 0;

**for** (**int** i = 0; i < (a.length); i++) {

**for** (**int** j = i + 1; j < a.length; j++) {

**if** (a[i] > a[j]) {

t = a[i];

a[i] = a[j];

a[j] = t;

}

}

System.***out***.println(a[i]);

}

System.***out***.println("second least salary is :"+a[1]);

}

}

44 .how to delete the value from the array which is modulus by 3 in JAVAScript

**var** a = [ 10, 9, 3, 2, 5 ];

**for** (**var** i = 0; i < a.length; i++) {

**if**(i===3){

**delete** a[i];

}

}

**var** b = [ 10, 9, 3, 2, 5 ];

b.splice(3,1); //Array.splice(index,length)

console.log('a is ',a,'and a[3] is ',**typeof** a[3]);

console.log('b is ',b);

// O/p a is [ 10, 9, 3, , 5 ] and a[3] is undefined

// b is [ 10, 9, 3 ]

Using **delete** , the particular index element got deleted and at that place undefined will be placed.

Using **splice** elements get deleted and memory is allocated to next index element ..i.e length decreases of array

45. How to add a property in JSON object at second position

Using splice method we can also add a property at any position

In bellow example a is an object

Object.splice(position, no. of elements to delete , element name to add)

a.splice(2,0,'we');

Object.prototype

JavaScript prototype property also allows you to add new methods to an existing prototype:

function Person(first, last, age, eyecolor) {  
    this.firstName = first;  
    this.lastName = last;  
    this.age = age;  
    this.eyeColor = eyecolor;  
}  
Person.**prototype**.nationality = "English";

Difference between (with .prototype and without)

Person.**prototype**.nationality = "English"; and

Person.nationality = "English";

Object.prototype will create an inheritance . Means if a property is added to an object with using .prototype then we can access that added property from any other module call.

But without using .prototype we can access that added property in the same module only and not outside.

46. ECMA script 6

**ECMAScript** (or ES) is a trademarked scripting-language specification standardized by Ecma International in ECMA-262 and ISO/IEC 16262. It was based on JavaScript, which now tracks **ECMAScript**. It is commonly used for client-side scripting on the World Wide Web.

**ECMAScript 6**, also known as **ECMAScript** 2015, is the latest version of the**ECMAScript** standard.

47. How to delete the array item and release the space too

Using **splice method**

48. What is exception?

An **exception** is an error that occurs at runtime due to an illegal operation during execution.

49. How exceptions are handled in node js

**Try**{

fs.open(“file.txt”, function (contents) {

console.log(contents);

});

}**Catch** (Error err) {

console.error(“An error occurred!”, err);

}

Try/catch will only catch exceptions thrown synchronously. The I/O error here will not be handled.

**50 How Node.js handles exceptions**

When a function in Node is called, it is typically passed a callback argument which is executed when the operation completes. This callback pattern is ubiquitous across Node.js applications because it is harmonious with the event loop.

The most common way of asynchronously signaling an exception is to pass the error object in as the*first argument* of the callback. The core Node.js API is designed to follow this convention and most third-party libraries also follow it.

Following our example above, let’s implement error handling the asynchronous way. Now we can check if an error has been produced in the callback itself.

fs.open(“file.txt”, function (**err,** contents) {

if (**err**) {

console.error(“An error occurred!”, err);

} else {

console.log(contents);

}

});

**51 How to handle the "Unhandled exceptions" in Node.js?**  
It can be caught at the "Process level" by attaching a handler for uncaughtException event.  
Example:

process.on('uncaughtException', function(err) {

console.log('Caught exception: ' + err);

});

52. Threshold in node and java

53. What is Express?

Express is a light-weight web application framework for web and mobile applications.

Express provides some additional features to extend server-side coding. It is better to use express rather than plain node.js for fast and scalable APIs.

Many frameworks are written on top of express (example: loopback)

**54. What is loopback?**

LoopBack is an open source Node.js framework built on top of Express.

**Express Cons**

* All end points need to be created manually, you end up doing a lot of the same code (or worse, start rolling your own libraries after a while)
* Every end point needs to be tested (or at the very least I recommend that you hit the end points with HTTP consumer to make sure they are actually there and don’t throw 500s)
* Refactoring becomes painful because everything needs to be updated everywhere
* Doesn’t come with anything “standard”, have to figure out your own approach

**Loopback Pros**

* Very quick RESTful API development
* Convention over configuration
* Built in models ready to use
* RPC support
* Fully configurable when needed
* Extensive documentation
* Fulltime team working on the project
* Available commercial support

55 What is strongloop?

Strongloop is the (IBM) company that has built an API Platform which features the open source Loopback framework. Loopback enables you to quickly compose APIs and runs on top of the express framework. It could've been named Strongloop Loopback Starter.

See: <https://strongloop.com/node-js/loopback-framework/>

In addition to the loopback framework, the Strongloop API Platform also includes the Arc graphical UI, which has tools for building, profiling, and monitoring Node Apps.

You could create your API using Loopback & then monitor & profile that API using Arc. Both are a part of the strongloop platform.

# **56. What is** [**slc arc**](https://docs.strongloop.com/display/NODE/slc+arc)

It’s a GUI for all strongloop products

your application will launched as a node cluster(having parent child process ) for production

**StrongLoop** Arc is a graphical tool for building, deploying, managing, and monitoring **LoopBack** applications and APIs. It enables you to: Create and modify **LoopBack** models and datasources.

Run [StrongLoop Arc](https://docs.strongloop.com/display/APIS/Using+Arc), by default opening it in a web browser window

Slc loopback

Give app name

Cd app directory

Here run slc arc

Go ro browser and use this GUI to create modules and other stuff.

StrongLoop Arc will use a different port number each time you run it. To run it on a specific port, use the PORT environment variable, for example:

|  |
| --- |
| $ PORT=4000 slc arc |

[*https://www.youtube.com/watch?v=ckD8KlZMl0k*](https://www.youtube.com/watch?v=ckD8KlZMl0k)

57. Folder structure of strongloop app.

Client

Common --models

Node\_modules

Server - boot

Package.json

Readme.md

-🡪 when slc arc is used one more folder is created [ .strong-pm]

**strong-pm** is a process manager that manages cluster size.

slc:is application generator

58.Diff bet slc run and node . to run whole application

Node . is used for development mode -- will launch only one process

Slc run is used by production mode.

59 .Authentication?

[*http://v2.wp-api.org/guide/authentication/*](http://v2.wp-api.org/guide/authentication/)

* Are you a plugin/theme running on the site? Use **cookie authentication**
* Are you a desktop/web/mobile client accessing the site externally? Use **OAuth authentication**, **application passwords**, or **basic authentication**

SON Web Tokens are an open, industry standard [**RFC 7519**](https://tools.ietf.org/html/rfc7519) method for representing claims securely between two parties.

JWT.IO allows you to decode, verify and generate JWT.

Passport is **authentication** middleware for **Node**.**js**

60. Versions using in project

Node v0.10.38

Strongloop v5.0.1

Loopback 2.22.1

Postman 4.7.0

Putty 0.66 latest is 0.67

Git 1.9.5 latest is 2.10.0

soapUI 4.6.1

parasoft 9.5

RAD 7.5.5

(Rational Application developer)

WAS 6.1

Eclipse LUNA

Eclipse types (LUNA, Mars, Neon, and Oxygen etc.)

**61. How to write a function in javaScript**

//1.

function sayHello(p1,p2) {

console.log("function demo hello");

}

//2.

var hii = function sayHii(p1,p2) {

console.log("function demo hii");

}

**Callback function**

App.get(‘\customers’,**function(err,res){**

**Console.log(‘this is in callback’);**

**}**);

**62 How to create an array in javaScript**

var cars = new Array("Saab", "Volvo", "BMW");

var cars = ["Saab", "Volvo", "BMW"];

var cars = []; ===array

var carObj = {}; == object

**63. Git commands**

1. git clone url.git

2. cd <dir>

3. git checkout develop

4. git checkout -b <give a good name and follow convention that Ned shared> develop ======this is for new branch creation

4. git checkout ngp2Iteration1NodeAPI

<add files/update/modify files>

6. git status

7. git add .

8. git commit –a [This will open vi editor. Update with proper comments]

9. git push origin ngp2Iteration1NodeAPI:ngp2Iteration1NodeAPI

1. Configure the author name and email address to be used with your commits.

Note that Git strips some characters (for example trailing periods) from user.name.

git config --global user.name "Sam Smith"

git config --global user.email sam@example.com

2.Create a new local repository

git init

3.Check out a repository

Create a working copy of a local repository:

git clone /path/to/repository

For a remote server, use:

git clone username@host:/path/to/repository

4.Add files

Add one or more files to staging (index):

git add <filename>

git add \*

5.Commit changes to head (but not yet to the remote repository):

git commit -m "Commit message"

6.Commit any files you've added with git add, and also commit any files you've changed since then:

git commit -a

7.Push :Send changes to the master branch of your remote repository:

git push origin master

8.Status : List the files you've changed and those you still need to add or commit:

git status

9.Connect to a remote repository

If you haven't connected your local repository to a remote server, add the server to be able to push to it:

git remote add origin <server>

10.List all currently configured remote repositories:

git remote -v

11.Branches :Create a new branch and switch to it:

git checkout -b <branchname>

12.Switch from one branch to another:

git checkout <branchname>

13.List all the branches in your repo, and also tell you what branch you're currently in:

git branch

14.Delete the feature branch:

git branch -d <branchname>

15.Push the branch to your remote repository, so others can use it:

git push origin <branchname>

16.Push all branches to your remote repository:

git push --all origin

17.Delete a branch on your remote repository:

git push origin :<branchname>

**64. What is use of app.use**

The app object is instantiated on creation of the Express server.

The app is an instance of express

To setup your middleware, you can invoke [app.use(<specific\_middleware\_layer\_here>)](http://expressjs.com/4x/api.html" \l "app.use) for every middleware layer that you want to add (it can be generic to all paths, or triggered only on specific path(s) your server handles), and it will add onto your [Express](http://expressjs.com/guide.html) middleware stack. Middleware layers can be added one by one in multiple invocations of use, or even all at once in series with one invocation.

app.use(cookieParser());

app.use(bodyParser());

**app.use is used to add or invoke middleware to express stack**

Express provides us app.use() method which is specifically used to define middlewares

**65.Types of middlewares in node – (e.g body-parser ?)**

<http://expressjs.com/en/guide/using-middleware.html>

https://www.youtube.com/watch?v=6z-hi2CF3c8

Middleware :

Middleware ensures the processing of some particular thing before the final request is being processed.

Express is a routing and middleware web framework

Middleware functions can perform the following tasks:

* Execute any code.
* Make changes to the request and the response objects.
* End the request-response cycle.
* Call the next middleware function in the stack.

bodyParser(); -Parse HTTP request body

cookieParser -Parse cookie header and populate req.cookies

multer - Handle multi-part form data.

[cookie-session](https://expressjs.com/en/resources/middleware/cookie-session.html) - Establish cookie-based sessions.

### **Route Handler**

http://qnimate.com/express-js-middleware-tutorial/

app.all(), app.get(), app.post(), app.delete() and app.put() methods are all used to define routes

There can be multiple route handlers executed for a single HTTP request. Here is an example:

var app = require("express")();  
  
app.get("/", function(httpRequest, httpResponse, next){  
    httpResponse.write("Hello");  
    next();  
});  
  
app.get("/", function(httpRequest, httpResponse, next){  
    httpResponse.write("World !!!");  
    httpResponse.end();  
});  
  
app.listen(8080);

output : **HelloWorld !!!**

Here the first handle writes some response and then calls next(). The next() method is used to call the next route handler match the route path.

**A route handler must end the request or call the next route handler.**

We can also pass multiple route handlers to a single call to app.all(), app.get(), app.post(), app.delete() and app.put() methods. Here is an example to demonstrate this:

var app = require("express")();  
  
app.get("/", function(httpRequest, httpResponse, next){  
    httpResponse.write("Hello");  
    next();  
}, function(httpRequest, httpResponse, next){  
    httpResponse.write("World !!!");  
    httpResponse.end();  
});  
  
app.listen(8080);

**Difference between app.use() and app.all():**

app.use() takes only one callback whereas app.all() can take multiple callbacks.

app.use() only see whether url starts with specified path where app.all() will match complete path.

Here is an example to demonstrate this:

app.use( "/product" , mymiddleware);  
*// will match /product*  
*// will match /product/cool*  
*// will match /product/foo*  
  
app.all( "/product" , handler);  
*// will match /product*  
*// won't match /product/cool   <-- important*  
*// won't match /product/foo    <-- important*  
  
app.all( "/product/\*" , handler);  
*// won't match /product        <-- Important*  
*// will match /product/cool*  
*// will match /product/foo*

response.write(" World !!!");

response.end();

Res.write will not end till you provide end to it. If you won’t give end() ; in browser it will be loading….loading …

response.send(" World !!!");

.send() doesn’t need end

**66.What is EJS templets etc**

### **EJS -** [**Embedded JavaScript templates**](https://www.google.co.in/url?sa=t&rct=j&q=&esrc=s&source=web&cd=3&sqi=2&ved=0ahUKEwih6JKu1KHVAhVJ1GMKHSMUDt8QFggpMAI&url=http%3A%2F%2Fejs.co%2F&usg=AFQjCNGJj1QqCdRpVlPfOqGdZDUrOe42BA)

 is a simple templating language that lets you generate HTML markup with plain JavaScript.

(This is to display the output came from database on the browser with some proper format .e.g create table OR list )

A **template engine** enables you to use static template files in your application. At runtime, the template engine replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client. This approach makes it easier to design an HTML page.

Some popular template engines that work with Express are [Pug](https://pugjs.org/api/getting-started.html), [Mustache](https://www.npmjs.com/package/mustache), and [EJS](https://www.npmjs.com/package/ejs). The [Express application generator](https://expressjs.com/en/starter/generator.html) uses [Jade](https://www.npmjs.com/package/jade) as its default, but it also supports several others.

 Jade has been renamed to [Pug](https://www.npmjs.com/package/pug).

To render template files, set the following [application setting properties](https://expressjs.com/en/4x/api.html#app.set), set in app.js in the default app created by the generator:

* views, the directory where the template files are located. Eg: app.set('views', './views'). This defaults to the views directory in the application root directory.
* view engine, the template engine to use. For example, to use the Pug template engine: app.set('view engine', 'pug').

Then install the corresponding template engine npm package; for example to install Pug:

npm install ejs --save

app.set('view engine', 'ejs')

Now Let’s first create an index.ejs file within the views folder so we can start populating data.

<ul class="quotes">

<% for(var i=0; i<quotes.length; i++) {%>

<li class="quote">

<span><%= quotes[i].name %></span>

<span><%= quotes[i].quote %></span>

</li>

<% } %>

</ul>

In EJS, you can write JavaScript within <% and %> tags. You can also output JavaScript as strings if you use the <%= and %> tags.

Once the view engine is set, we can begin generating the HTML with our quotes. This process is also called **rendering**. We can use the render object built into the response object render to do so. It has the following syntax:

res.render(views, locals)

**The first parameter, views**, is the name of the file we’re rendering. This file must be placed within a views folder.

**The second parameter, locals**, is an object that passes data into the view.

app.get('/', (req, res) => {

db.collection('quotes').find().toArray((err, result) => {

if (err) return console.log(err)

// renders index.ejs

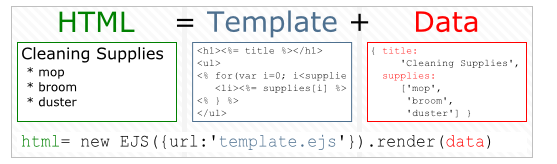
res.render('index.ejs', {quotes: result})

})

})

**67 What is rendering in node js**

res.render() function compiles your template inserts locals(data) there, and creates html output out of those two things.



The render method works when you have a templating engine in use such as handlebars.js or jade OR EJS.

A templating engine is a node module associated with express (which some people refer to as an express plugin) which parses the template file and generate the HTML output.

The **sendfile** method simply sends the file to the client.

Since you are using an HTML file, there is nothing particularly to be parsed by the templating engine. So, the output of render is same as that of sendfile (i.e., the HTML written in the file). Hence, both produce the same result.

app.get('/',function(req,res){  
  res.**sendFile**('index.html');  
  *//It will find and locate index.html from View or Scripts*  
});

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

res.**render**('index.html');

});

**68 How Routing is done in node (URL routing)**

**Routing** refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).

Each route can have one or more handler functions, which are executed when the route is matched.

Route definition takes the following structure:

app.METHOD(PATH, HANDLER)

Where:

* app is an instance of express.
* METHOD is an [HTTP request method](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods), in lowercase.(get,put,post,delete)
* PATH is a path on the server.(“/myPage”)
* HANDLER is the function executed when the route is matched.

The following examples illustrate defining simple routes.

Respond with Hello World! on the homepage:

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

res.send('Hello World!')

})

Respond to POST request on the root route (/), the application’s home page:

app.post('/', function (req, res) {

res.send('Got a POST request')

})

69 Connectors for different data sources and which we are using?

* Mongodb

'mongodb' 🡪 Connector for express

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/mydb";  
  
**MongoClient.connect**(url, function(err, db) {  
  if (err) throw err;  
  console.log("Database created!");  
  db.close();  
});

'loopback-connector-mongodb ' 🡪 Connector for strongloop

* Mysql

'mysql' 🡪 Connector for express

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword"  
});  
  
con.connect(function(err) {  
 if (err) throw err;  
  console.log("Connected!");  
  con.query("**CREATE DATABASE mydb**", function (err, result) {  
    if (err) throw err;  
    console.log("Database created");  
  });  
});

'loopback-connector-mysql ' 🡪 Connector for strongloop

70 CRUD (Create Read Update Delete)

### 71 How Many Types Of Streams Are Present In Node.Js?

**Answer.**

Stream in Node.js are objects that allow reading data from a source or writing data to a specific destination in a continuous fashion. In Node.js, there are four types of streams.

* **<Readable> –** This is the Stream to be used for reading operation.
* **<Writable> –** It facilitates the write operation.
* **<Duplex> –** This Stream can be used for both the read and write operations.
* **<Transform> –** It is a form of a duplex Stream, which performs the computations based on the available input.

All the Streams, discussed above are an instance of an “EventEmitter” class. The event thrown by the Stream varies with time. Some of the commonly used events are as follows.

* **<data> –** This event gets fired when there is data available for reading.
* **<end> –** The Stream fires this event when there is no more data to read.
* **<error> –** This event gets fired when there is any error in reading or writing data.
* **<finish> –** It fires this event after it has flushed all the data to the underlying system.
  1. **What Is EventEmitter In Node.Js?**
* **Answer.**
* Events module in Node.js allows us to create and handle custom events. The Event module contains “EventEmitter” class which can be used to raise and handle custom events. It is accessible via the following code.
* JavaScript
* 

|  |  |
| --- | --- |
| 1  2  3  4  5 | // Import events module  var events = require('events');    // Create an eventEmitter object  var eventEmitter = new events.EventEmitter(); |

* When an EventEmitter instance encounters an error, it emits an “error” event. When a new listener gets added, it fires a “newListener” event and when a listener gets removed, it fires a “removeListener” event.
* EventEmitter provides multiple properties like “on” and “emit”. The “on” property is used to bind a function to the event and “emit” is used to fire an event.

### 73 What Is The Difference Between Nodejs, AJAX, And JQuery?

**Answer.**

The one common trait between Node.js, AJAX, and jQuery is that all of them are the advanced implementation of JavaScript. However, they serve completely different purposes.

#### Node.Js –

It is a server-side platform for developing client-server applications. For example, if we’ve to build an online employee management system, then we won’t do it using client-side JS. But the Node.js can certainly do it as it runs on a server similar to Apache, Django not in a browser.

#### AJAX (Asynchronous Javascript And XML) –

It is a client-side scripting technique, primarily designed for rendering the contents of a page without refreshing it. There are a no. of large companies utilizing AJAX such as Facebook and Stack Overflow to display dynamic content.

#### JQuery –

It is a famous JavaScript module which complements AJAX, DOM traversal, looping and so on. This library provides many useful functions to help in JavaScript development. However, it’s not mandatory to use it but as it also manages cross-browser compatibility, so can help you produce highly maintainable web applications.

**74 What is nodemon**

**Nodemon restarts the server automatically** whenever you save a file that the server uses. We can install Nodemon by using the following command:

$ npm install **nodemon** --save-dev

Note: The reason we’re using a --save-dev flag here is because we’re only using Nodemon when we’re developing. This flag would save Nodemon as a devDependency in your package.json file.

Moving on, Nodemon behaves exactly the same as node, which means we can run our server by calling nodemon server.js. However, we can’t do it in the command line right now because Nodemon isn’t installed with a -g flag.

There’s one other way to run Nodemon – we can execute Nodemon from the node\_modules folder. The code looks like this:

$ ./node\_modules/.bin/nodemon server.js

**74 Closure**

**75 prototype , JavaScript inheretance**

**76 Promises**

**77 callback and callback hell how to avoid**

**78 what is need to use streams –**

To get data in chunks

**79 what is mongoose in node**

[Mongoose](http://mongoosejs.com/) is an object data modeling (ODM) library that provides a rigorous modeling environment for your data, enforcing structure as needed while still maintaining the flexibility that makes MongoDB powerful.

Mongoose provides a straight-forward, schema-based solution to model your application data. It includes built-in type casting, validation, query building, business logic hooks and more, out of the box.

Mongoose allows us to have access to the MongoDB commands for CRUD simply and easily.

**OAuth 2.0**

**Bearer Token – Is OAuth 2.0 subpart**

Used to access protected resources

OAuth 2.0 - Accessing a Protected Resource

They are included in the authorization header as follows −

Authorization: Bearer [token-value]

OAuth 2.0 is a very flexible protocol that relies on SSL (Secure Sockets Layer that ensures data between the web server and browsers remain private) to save user access token.

OAuth 2.0 is a simple protocol that allows to access resources of the user without sharing passwords.

It allows sharing of resources stored on one site to another site without using their credentials.

Following are the parameters and their descriptions.

* **client\_id** − It should be set to the client id of your application.
* **redirect\_uri** − It should be set to the URL. After the request is authorized, the user will be redirected back.
* **response\_type** − It can either be a code or a token. The code must be used for server side applications, whereas the token must be used for client side applications. In server side applications, you can make sure that the secrets are saved safely.
* **We use scope also**

https://publicapi.example.com/oauth2/authorize?client\_id=your\_client\_id&redirect\_uri=your\_url

&response\_type=code



**HTML DOM -**

**Document Object Model** is a wayto access and modify the document.

When a web page is loaded, the browser creates a **D**ocument **O**bject **M**odel of the page

The Objects are organized in a hierarchy. Top window object , then Doument object and then its childs.

The Document Object Model is a programming API for documents. The object model itself closely resembles the structure of the documents it models. For instance, consider this table, taken from an HTML document:

<TABLE>

<ROWS>

<TR>

<TD>Shady Grove</TD>

<TD>Aeolian</TD>

</TR>

<TR>

<TD>Over the River, Charlie</TD>

<TD>Dorian</TD>

</TR>

</ROWS>

</TABLE>

The Document Object Model represents this table like this:

