1. **When to use PUT and POST methods in REST?**

**Ans.** There is also another theory which says that for creating new things, you should use PUT if the unique identifier is provided by client i.e. client is responsible for creating entity e.g. client can create resource /user/joe/ by providing username joe and that would be unique URI. Similar, use POST if the server is responsible for creating new resources e.g. if ID is part of URI and automatically created by the server.

1. **Get parameter in node js?**

**Ans.**

app.get('/users/:userId/books/:bookId', function (req, res) {  
 res.send(req.params)  
})

1. **Serving static files in Express ?**

Ans. app.use(express.static('public'))

1. **Ejs –**

Ans. <h1><%= title %></h1>   
<p>Welcome to <%= title %></p>   
<% for(var i=0; i<supplies.length; i++) {%>   
<% if(i==1) { %> <span onclick='testFun()'> loop is <%= i %> position </span> <%}%>   
<br>   
<span><%= supplies[i] %></span>  
<% } %>  
<form action="/ejsform" method="get">   
 <label for="team\_name">Enter name: </label>   
<input id="team\_name" type="text" name

Controller - let data = { title: 'Express',

supplies: ['mop', 'broom', 'duster']

}

res.render('index', data);

1. **Node authentication and authorization**

Ans. Authentication – passport.js

Authorization -npm install acl

1. **Rest parameter and spread operator?**

Ans.

**Rest parameter-**

function doMath(operator, ...numbers) {

console.log(operator);

console.log(numbers);

}

doMath('add', 1, 2, 3);

**Spread operator –**

function add(...numbersToAdd) { // This is a Rest parameter

return numbersToAdd.reduce((sum, next) => sum + next);

}

var numbers = [1, 2, 3];

add(...numbers); // this is a Spread operator

// The above is functionally the same as:

add(1, 2, 3);

1. **Singleton design pattern?**

Ans. The getInstance method is Singleton's gatekeeper. It returns the one and only instance of the object while maintaining a private reference to it which is not accessible to the outside world.

**Singleton with a closure-**

function User() {

// the cached instance

var instance;

// rewrite the constructor

User = function() {

return instance;

};

// carry over the prototype

User.prototype = this;

// the instance

instance = new User();

// reset the constructor pointer

instance.constructor = User;

// all the functionality

instance.firstName = 'John';

instance.lastName = 'Doe';

return instance;

}

**Singleton with a self executing function-**

var User;

(function() {

var instance;

User = function User() {

if (instance) {

return instance;

}

instance = this;

// all the functionality

this.firstName = 'John';

this.lastName = 'Doe';

return instance;

};

}());

1. **if(NaN == NaN) {**

Ans. false

1. **type of NaN?**

Ans. Number.

1. **Model In sequelize?**

Ans. const Task = sequelize.define('task', {

title: Sequelize.STRING,

description: Sequelize.TEXT,

deadline: Sequelize.DATE

})

Model.findAll({

attributes: ['foo', 'bar'],

where: {

authorId: 2

}

});

Post.findAll({

where: {

authorId: 2

}

});

**11 – sequelize query and association and model?**

**Ans-** const Player = this.sequelize.define('player', {/\* attributes \*/});

const Team = this.sequelize.define('team', {/\* attributes \*/});

Player.belongsTo(Team);

1. **Prototype -**

Ans. function talk() {

console.log(this.sound)

}

let animal = {

talk

}

let cat = {

sound: "meow"

}

Object.setPrototypeOf(cat, animal)

**Below will create new prototype:**

const food = {

init:function(type) {

this.type

},

eat:function() {

console.log('you are the' +this.type)

}

}

const waffle = object.create(food);

const correct = object.create(food)

waffle.init('waffle')

waffle.eat()

carrot.init('corrot')

corrot.eat()

1. **Copy One table to another table from another server?**

Ans . SELECT \* INTO targetTable FROM [sourceserver].[sourcedatabase].[sourcetable]

1. **multer in node js?**

**Ans.** var storage = multer.diskStorage({

destination: function (req, file, cb) {

cb(null, 'uploads/')

},

filename: function (req, file, cb) {

cb(null, file.fieldname + '-' + Date.now()+'.jpg')

}

})

var upload = multer({ storage: storage }).single('avatar');

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

console.log('aa');

upload(req, res, function (err) {

if (err) {

// An error occurred when uploading

return

}

res.json({messge:'done'})

// Everything went fine

})

})

1. **Recursion?**

Ans. function fact(x) {

if(x==0) {

return 1;

}

return x \* fact(x-1);

}

function run(number) {

alert(fact(parseInt(number, 10)));

}

1. **variable hosting?**

Ans. Hoisting is JavaScript's default behavior of moving all declarations to the top of the current scope (to the top of the current script or the current function).

Variables and constants declared with letor const are not hoisted!

1. **clone the object?**

Ans. var clone = Object.assign({}, obj);

1. **lexical scoping?**

Ans. Lexical Scoping First off, JavaScript has lexical scopingwith function scope. In other words, even though JavaScript looks like it should have block scope because it uses curly braces { }, a new scope is created only when you create a new function.

var outerFunction = function(){

if(true){

var x = 5;

//console.log(y); //line 1, ReferenceError: y not defined

}

var nestedFunction = function() {

if(true){

var y = 7;

console.log(x); //line 2, x will still be known prints 5

}

if(true){

console.log(y); //line 3, prints 7

}

}

return nestedFunction;

}

var myFunction = outerFunction();

myFunction();

In this example, the variable x is available everywhere inside of outerFunction(). Also, the variable y is available everywhere within the nestedFunction(), but neither are available outside of the function where they were defined. The reason for this can be explained by lexical scoping. The scope of variables is defined by their position in source code. In order to resolve variables, JavaScript starts at the innermost scope and searches outwards until it finds the variable it was looking for. Lexical scoping is nice, because we can easily figure out what the value of a variable will be by looking at the code; whereas in dynamic scoping, the meaning of a variable can change at runtime, making it more difficult.

1. **Scope chaining?**

Ans. Once the function has finished executing, its context is popped off the stack. Within each execution context is a special object called a scope chain which is used to resolve variables. A scope chain is essentially a stack of currently accessible scopes, from the most immediate context to the global context.

function world() {

var name = 'global';

(function() {

var name = 'country';

alert(name)

})();

alert(name)

}

**Output would be contury and global**

1. **promises chaining**

Ans. let cleanRoom = function() {

return new Promise(function(resolve, reject) {

resolve('clean the Room');

})

}

let removeGarbage = function(message) {

return new Promise(function(resolve, reject){

resolve(message)

})

}

cleanRoom().then(function(result) {

return removeGarbage(result);

}).then(function(result1){

console.log(result1)

})

1. **Scoping?**

Ans. In JavaScript there are two types of scope:

1. Local scope
2. Global scope
3. **context?**

Ans. scope are for function context for object

Context is related to objects. It refers to the object to which a function belongs. When you use the JavaScript “this” keyword, it refers to the object to which function belongs.

Scope refers to the visibility of variables, and content refers to the object to which a function belongs.

For example, inside of a function, when you say: “this.accoutNumber”, you are referring to the property “accoutNumber”, that belongs to the object to which that function belongs. If the object “foo” has a method called “bar”, when the JavaScript keyword “this” is used inside of “bar”, it refers to “foo”. If the function “bar” were executed in the global scope, then “this” refers to the window object (except in strict mode). It is important to keep in mind that by using the JavaScript call() or apply() methods, you can alter the context within which a function is executed. This, in-turn, changes the meaning of “this” inside of that function when it is executed.

1. **this?**

Ans. In most cases, the value of this is determined by how a function is called. It can't be set by assignment during execution, and it may be different each time the function is called. ES5 introduced the bind method to set the value of a function's this regardless of how it's called, and ES2015 introduced arrow functions which don't provide their own this binding (it retains the this value of the enclosing lexical context).

1. **remove last element?**

Ans. fruits.pop();

1. **error first**

Ans. With this pattern, a callback function is passed to the method as an argument. When the operation either completes or an error is raised, the callback function is called with the Error object (if any) passed as the first argument. If no error was raised, the first argument will be passed as null .

1. **how to skip the file from nodemon?**

Ans. {

"verbose": true,

"ignore": ["\*.test.js", "fixtures/\*"],

"execMap": {

"rb": "ruby",

"pde": "processing --sketch={{pwd}} --run"

}

}

1. **Error Handling –**

Ans. **The default error handler-**

Express comes with a built-in error handler that takes care of any errors that might be encountered in the app. This default error-handling middleware function is added at the end of the middleware function stack.

If you pass an error to next() and you do not handle it in a custom error handler, it will be handled by the built-in error handler; the error will be written to the client with the stack trace. The stack trace is not included in the production environment.

function errorHandler (err, req, res, next) {

if (res.headersSent) {

return next(err)

}

res.status(500)

res.render('error', { error: err })

}

1. **JWT –**

Ans. jsonwebtoken

npm install jsonwebtoken -

Signing a token with 1 hour of expiration:

jwt.sign({

data: 'foobar'

}, 'secret', { expiresIn: 60 \* 60 });

jwt.verify(token, secretOrPublicKey, [options, callback])

jwt.verify(token, 'shhhhh', function(err, decoded) {

console.log(decoded.foo) // bar

});

jwt.decode(token [, options])

var decoded = jwt.decode(token, {complete: true});

1. **Child process –**

Ans. The cluster module provides a way of creating child processes that runs simultaneously and share the same server port.

Node.js runs single threaded programming, which is very memory efficient, but to take advantage of computers multi-core systems, the Cluster module allows you to easily create child processes that each runs on their own single thread, to handle the load.

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();

}

function childProcess() {

console.log(`Worker ${process.pid} started and finished`);

process.exit();

}

1. Pm2 module -

Ans - PM2 is a Production Runtime and Process Manager for Node.js applications with a built-in Load Balancer. It allows you to keep applications alive forever, to reload them without downtime and facilitate common Devops tasks.

Starting an application in production mode is as easy as:

pm2 start app.js

To list all running applications:

pm2 list

Managing apps is straightforward:

$ pm2 stop <app\_name|id|'all'|json\_conf>

$ pm2 restart <app\_name|id|'all'|json\_conf>

$ pm2 delete <app\_name|id|'all'|json\_conf>

To have more details on a specific application:

pm2 describe <id|app\_name>

To monitor logs, custom metrics, application information:

pm2 monit

Hot Reload allows to update an application without any downtime:

pm2 reload all

1. **What are Streams?**

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

Readable − Stream which is used for read operation.

Writable − Stream which is used for write operation.

Duplex − Stream which can be used for both read and write operation.

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

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

data − This event is fired when there is data is available to read.

end − This event is fired when there is no more data to read.

error − This event is fired when there is any error receiving or writing data.

finish − This event is fired when all the data has been flushed to underlying system.

Reading from a Stream

var fs = require("fs");

var data = '';

// Create a readable stream

var readerStream = fs.createReadStream('input.txt');

// Set the encoding to be utf8.

readerStream.setEncoding('UTF8');

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

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

data += chunk;

});

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

console.log(data);

});

readerStream.on('error', function(err){

console.log(err.stack);

});

1. **buffer –**

Ans. Pure JavaScript is Unicode friendly, but it is not so for binary data. While dealing with TCP streams or the file system, it's necessary to handle octet streams. Node provides Buffer class which provides instances to store raw data similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap.

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

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

var buf = new Buffer("Simply Easy Learning", "utf-8");

1. **log maintaining –**

Ans. Winston, Bunyan, Child Loggers, Morgan,

1. **body parser –**

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

1. **Node js module?**

Ans.

1. Fs
2. Event
3. Cluster
4. Stream
5. Buffer
6. http
7. https
8. **Promise syntax?**

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

if(false) {

resolve('some Message')

}else {

reject('Some error')

}

});

promise.then(function(message) {

console.log(message)

}, function(message){

console.log(message)

})

**37- break in foreach**

Ans. break;

**38- Difference between function and arrow function**

Ans. Arrow functions don't have their own this or arguments binding. Instead, those identifiers are resolved in the lexical scope like any other variable. That means that inside an arrow function, this and arguments refer to the values of this and arguments in the environment the arrow function is defined in (i.e. "outside" the arrow function):

function createObject() {

console.log('Inside `createObject`:', this.foo);

return {

foo: 42,

bar: function() {

console.log('Inside `bar`:', this.foo);

},

};

}

createObject.call({foo: 21}).bar(); // override `this` inside createObject

**output-**

Inside `createObject`: 21

Inside `bar`: 42

**Arrow function-**

/ Example using a arrow function

function createObject() {

console.log('Inside `createObject`:', this.foo);

return {

foo: 42,

bar: () => console.log('Inside `bar`:', this.foo),

};

}

createObject.call({foo: 21}).bar(); // override `this` inside createObject

**output**

Inside `createObject`: 21

Inside `bar`: 21

**40-Multithreaded and multi process node js?**

Ans. Node.js is a single threaded language which in background uses multiple threads to execute asynchronous code.

Node.js is non-blocking which means that all functions ( callbacks ) are delegated to the event loop and they are ( or can be ) executed by different threads. That is handled by Node.js run-time.

Node.js does support forking multiple processes ( which are executed on different cores ).

It is important to know that state is not shared between master and forked process.

We can pass messages to forked process ( which is different script ) and to master process from forked process with function send.

Why and when we need to fork another process?

Forking multiple processes is essential for freeing up memory and unloading single process.

When we need to delegate tasks ( run them in parallel ) to another process for the sake of the speed.

**38-i f [] if {}**

Ans. true, true

**39- api case sensitive, get post increment on value**

Ans. api not case sensitive and value cannot be incremented

**40 - What are mutable and immutable variables?**

Ans. A mutable object is an object whose state can be modified after it is created. An immutable object is an object whose state cannot be modified after it is created. Examples of native JavaScript values that are immutable are numbers and strings. Examples of native JavaScript values that are mutable include objects, arrays, functions, classes, sets, and maps.

**41- variables scope**

Ans. var outside function - not defined

function abc1() {  
      
    var ty = 1;  
    callb();  
}  
abc1();  
console.log(ty);

--------------

var ty =21  
  
if(1==1) {  
    let ty =1;  
      
}  
  
console.log(ty);

---------------------------------------

let ty =21  
  
if(1==1) {  
    var ty =1;  
      
}  
  
console.log(ty);

42- inheritance?

Ans.

43- promise alternative?

Ans.

44- benefit of promise?

Ans. They, along with callbacks, streams, and generators/thunks, provide a way to deal with deferred work in your application. For example, returning the result of query to the client when it comes back from the database. The main advantage of a promise over a callback is that you have an instance of an object that represents that deferred work. Let's say for example that you wanted to do two things in parallel after some asynchronous action. With callbacks, you would have to create a new callback that managed that flow. With promises, you can get a promise from the first action and pass it to methods that do the next action and get promises back.

45- promise inside

Ans.

46- outside function in class?

Ans.

47- rest ans soap api

Ans. Ans. There are many differences between SOAP and REST web services. The important 10 differences between SOAP and REST are given below:

1. SOAP is a protocol. REST is an architectural style.
2. SOAP stands for Simple **Object Access Protocol**. REST stands for **REpresentational State Transfer**.
3. SOAP can't use REST because it is a protocol. REST can use SOAP web services because it is a concept and can use any protocol like HTTP, SOAP.
4. SOAP uses services interfaces to expose the business logic. REST uses URI to expose business logic.
5. In Java JAX-WS is the java API for SOAP web services. In Java JAX-RS is the java API for RESTful web services.
6. SOAP defines standards to be strictly followed. REST does not define too much standards like SOAP.
7. SOAP requires more bandwidth and resource than REST. REST requires less bandwidth and resource than SOAP.
8. SOAP defines its own security. RESTful web services inherits security measures from the underlying transport.
9. SOAP permits XML data format only. REST permits different data format such as Plain text, HTML, XML, JSON etc.
10. SOAP is less preferred than REST. REST more preferred than SOAP.

Difference between Rest and Soap

**SOAP:**

1. SOAP is a protocol.
2. SOAP stands for Simple Object Access Protocol.
3. SOAP can't use REST because it is a protocol.
4. SOAP uses services interfaces to expose the business logic.
5. SOAP defines standards to be strictly followed.
6. SOAP requires more bandwidth and resource than REST.
7. SOAP defines its own security.
8. SOAP permits XML data format only.
9. SOAP is less preferred than REST.

**REST**

1. REST is an architectural style.
2. REST stands for Representational State Transfer.
3. REST can use SOAP web services because it is a concept and can use any protocol like HTTP, SOAP.
4. REST uses URI to expose business logic.
5. REST does not define too much standards like SOAP.
6. REST requires less bandwidth and resource than SOAP.
7. RESTful web services inherit security measures from the underlying transport.
8. REST permits different data format such as Plain text, HTML, XML, JSON etc.
9. REST more preferred than SOAP.

49-  lamda function

Ans,

50- graphql?

Ans,

51- Apollo client?

Ans.

52- boiler plate and holder architecture?

Ans.

53- thread pool

Ans. threadpool-js

54- prototype chaining?

Ans. Prototype chaining is used to build new types of objects based on existing ones. It has a very similar job to inheritance in a class based language

Constructor functions have a property called prototype. Adding properties and methods to the prototype property will automatically add the method or property to all objects created by the constructor function.

Prototype chaining is an extension of this idea. The prototype property is just a regular javascript object so it’s possible to create a function’s prototype using another constructor function. When you do this, all of the properties and methods from the constructor function’s prototype are automatically added to new the prototype object. This makes it easy to create a constructor function that builds objects that are an extended version of an existing one.

For a simple example, imagine that you have a Pet constructor to make pet objects. Now you want to make Cat that is pretty much like a Pet but has a few differences.

Here’s the Pet constructor function from the prototype example:

function Pet(name, species, hello)

{

this.name = name;

this.species = species;

this.hello = hello;

}

Pet.prototype = {

sayHello : function()

{

alert(this.hello);

}

}

56- git

Ans,

57- fire base?

Ans,

58- unit testing?

Ans.

59- express validator

Ans. npm install express-validator

60- debugger

Ans,

61- heppi

Ans,

62- http and tcp

Ans. TCP works in the Transport layer while HTTP works in Application layer of TCP/IP model. This just means that HTTP works on top of TCP. TCP is in charge of setting up a reliable connection between two machines and HTTP uses this connection to transfer data between the server and the client. HTTP is used for transferring data while TCP is in charge of setting up a connection which should be used by HTTP in the communication process. Without TCP, HTTP cannot function (to be crisp).

63- log time in redis

Ans.

64-events

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

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.

// Import events module

var events = require('events');

// Create an eventEmitter object

var eventEmitter = new events.EventEmitter();

// Create an event handler as follows

var connectHandler = function connected() {

console.log('connection succesful.');

// Fire the data\_received event

eventEmitter.emit('data\_received');

}

// Bind the connection event with the handler

eventEmitter.on('connection', connectHandler);

// Bind the data\_received event with the anonymous function

eventEmitter.on('data\_received', function(){

console.log('data received succesfully.');

});

// Fire the connection event

eventEmitter.emit('connection');

console.log("Program Ended.");

**EventEmitter Class-**

As we have seen in the previous section, EventEmitter class lies in the events module. It is accessible via the following code −

65- closure?

Ans. 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.

The inner function has access not only to the outer function’s variables, but also to the outer function’s parameters. Note that the inner function cannot call the outer function’s arguments object, however, even though it can call the outer function’s parameters directly.

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

66- Array Methods & String Methods.

Ans. toString() - The JavaScript method toString() converts an array to a string of (comma separated) array values.

The join() - The join() method also joins all array elements into a string.

The pop() - The pop() method removes the last element from an array:

push() - The push() method adds a new element to an array (at the end):

push() - The push() method returns the new array length:

shift() -The shift() method removes the first array element and "shifts" all other elements to a lower index.

unshift() - The unshift() method adds a new element to an array (at the beginning), and "unshifts" older elements:

delete- Since JavaScript arrays are objects, elements can be deleted by using the JavaScript operator delete:

delete fruits[0];

splice() -The splice() method can be used to add new items to an array:

fruits.splice(2, 0, "Lemon", "Kiwi");

With clever parameter setting, you can use splice() to remove elements without leaving "holes" in the array:

fruits.splice(0, 1);

concat() -The concat() method can take any number of array arguments:

arr1.concat(arr2, arr3);

slice()- The slice() method slices out a piece of an array into a new array.

sort()-The sort() method sorts an array alphabetically:

reverse() - The reverse() method reverses the elements in an array.

Numeric Sort-

assending

var points = [40, 100, 1, 5, 25, 10];

points.sort(function(a, b){return a - b});

deccending-

var points = [40, 100, 1, 5, 25, 10];

points.sort(function(a, b){return b - a});

String methods-

txt.length; -The length property returns the length of a string:

str.indexOf("locate");- The indexOf() method returns the index of (the position of) the first occurrence of a specified text in a string:

str.lastIndexOf("locate");- The lastIndexOf() method returns the index of the last occurrence of a specified text in a string:

str.search("locate");- The search() method searches a string for a specified value and returns the position of the match:

str.slice(7, 13); - slice() extracts a part of a string and returns the extracted part in a new string.

str.substring(7, 13); - substr() is similar to slice().

str.substr(7, 6); -substr() is similar to slice().

str.replace("Microsoft", "W3Schools"); - The replace() method replaces a specified value with another value in a string:

text1.concat(" ", text2); - concat() joins two or more strings:

String.trim() - String.trim() removes whitespace from both sides of a string.

str.charAt(0); - The charAt() method returns the character at a specified index (position) in a string:

txt.split(","); - A string can be converted to an array with the split() method:

68- Yield

Ans. The yield keyword pauses generator function execution and the value of the expression following the yield keyword is returned to the generator's caller. It can be thought of as a generator-based version of the return keyword.

The yield keyword actually returns an IteratorResult object with two properties, value and done. The value property is the result of evaluating the yield expression, and done is false, indicating that the generator function has not fully completed.

Once paused on a yield expression, the generator's code execution remains paused until the generator's next() method is called. Each time the generator's next() method is called, the generator resumes execution and runs until it reaches one of the following:

A yield, which causes the generator to once again pause and return the generator's new value. The next time next() is called, execution resumes with the statement immediately after the yield.

throw is used to throw an exception from the generator. This halts execution of the generator entirely, and execution resumes in the caller as is normally the case when an exception is thrown.

The end of the generator function is reached; in this case, execution of the generator ends and an IteratorResult is returned to the caller in which the value is undefined and done is true.

A return statement is reached. In this case, execution of the generator ends and an IteratorResult is returned to the caller in which the value is the value specified by the return statement and done is true.

69- fibbonaci

Ans. 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 …

var fibonacci\_series = function (n)

{

if (n===1)

{

return [0, 1];

}

else

{

var s = fibonacci\_series(n - 1);

s.push(s[s.length - 1] + s[s.length - 2]);

return s;

}

};

console.log(fibonacci\_series(8));

[0,1,1,2,3,5,8,13,21]

70-factorial

Ans. 5! = 5 \* 4 \* 3 \* 2 \* 1

function factorialize(num) {

if (num < 0)

return -1;

else if (num == 0)

return 1;

else {

return (num \* factorialize(num - 1));

}

}

factorialize(5);

71- Difference between promise and async

Ans. Async/await is built on top of Promises. If a function returns a Promise, you can await the actual result. And an async function always returns a Promise.

Just do everything with Promises. Once you can use async/await, using those promise-returning functions will become much simpler, but you will still need to understand how Promises work.

So, it's not an either-or thing. Promises are the foundation and async/await is sitting on top.

72- why node js single thereded,

Ans. because javascript was signle threaded

73- load balancing,

Ans. There are plenty of powerful load balancing tools out there, like nginx or HAProxy. Nginx and HAProxy are fast and battle-tested, but can be hard to extend if you’re not familiar with C. Nginx has support for a limited subset of JavaScript, but nginScript is not nearly as sophisticated as Node.js. If you’re looking for a load balancer that you can extend with Node.js, look no further than Express, the most popular Node.js web framework. In this article, I’ll show you how to build your own load balancer with 10 lines of Express code, and show you how you can extend this load balancer to handle profiling and SSL termination.

**About Load Balancers & Express**

A load balancer is a process that takes in HTTP requests and forwards these HTTP requests to one of a collection of servers. Load balancers are usually used for performance purposes: if a server needs to do a lot of work for each request, one server might not be enough, but two servers alternating handling incoming requests might.

First off, let’s install express and request. The request package is an HTTP client with good support for streams, using it will make writing the load balancer very easy.

74- uppercase and lower case?

Ans. str.toUpperCase();

str.toLowerCase();

75- this, bind, apply

Ans. let obj = {'name':"john"};

let hello = function () {

console.log(this.name);

}

hello.call(obj, 1,1,3);

hello.apply(obj, [1,32,45]);

hello.bind(obj)();

77- constructor

Ans. The examples from the previous chapters are limited. They only create single objects.

Sometimes we need a "blueprint" for creating many objects of the same "type".

The way to create an "object type", is to use an object constructor function.

In the example above, function Person() is an object constructor function.

Objects of the same type are created by calling the constructor function with the new keyword:

function Person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

}

var myFather = new Person("John", "Doe", 50, "blue");

var myMother = new Person("Sally", "Rally", 48, "green");

78- thunkss

Ans. In computer programming, a thunk is a subroutine used to inject an additional calculation into another subroutine. Thunks are primarily used to delay a calculation until its result is needed, or to insert operations at the beginning or end of the other subroutine.

const thunk = require('thunks')()

const fs = require('fs')

* thunk is a function that encapsulates synchronous or asynchronous code inside.
* thunk accepts only one callback function as an arguments, which is a CPS function.
* thunk returns another thunk function after being called, for chaining operations.
* thunk passes the results into a callback function after being excuted.
* If the return value of callback is a thunk function, then it will be executed first and its result will be sent to another thunk for excution, or it will be sent to another new thunk function as the value of the computation.

thunk(function (done) {

fs.stat('package.json', done)

})(function (error, res) {

console.log(error, res)

})