**JavaScript:**

<https://www.fullstack.cafe/blog/javascript-code-interview-questions><https://hackr.io/blog/javascript-interview-questions>

<https://arc.dev/developer-blog/javascript-interview-questions/>

<https://www.codingame.com/playgrounds/9799/learn-solve-call-apply-and-bind-methods-in-javascript>

<https://blog.bitsrc.io/most-important-javascript-coding-challenge-aa14c956d2df>

<https://www.freecodecamp.org/news/how-to-fetch-data-from-an-api-using-the-fetch-api-in-javascript/>

const vs object.freeze

Object. freeze works on values, and more specifically, object values. It makes an object immutable, i.e. you cannot change its properties. Basically, const is the new var ; it's just block-scoped and prevents reassignment.

<https://www.testdome.com/tests/javascript-node-js-online-test/111>

<https://deepajarout.medium.com/when-do-we-have-to-use-apply-call-bind-bae0e896b6ec>

<https://deepajarout.medium.com/javascripts-basic-idea-of-hoisting-63612140da87>

<https://deepajarout.medium.com/difference-between-map-and-foreach-d17ae63337d4>

<https://deepajarout.medium.com/80-most-asked-javascript-interview-questions-tcs-accenture-ibm-infosys-etc-29cd395096bd>

<https://deepajarout.medium.com/most-asked-interview-question-object-freeze-vs-object-seal-c05bb366e613>

<https://www.linkedin.com/pulse/life-cycle-http-request-mamun-hoque>

<https://deepajarout.medium.com/what-is-the-second-parameter-in-parseint-1d93bb37b0b1>

<https://www.freecodecamp.org/news/copying-stuff-in-javascript-how-to-differentiate-between-deep-and-shallow-copies-b6d8c1ef09cd/>

<https://stackoverflow.com/questions/16109108/javascript-creating-objects-multiple-approaches-any-differences>

<https://www.freecodecamp.org/news/the-difference-between-arrow-functions-and-normal-functions/>

<https://www.freecodecamp.org/news/javascript-rest-vs-spread-operators/>

<https://medium.com/@dylanguidry20/a-comparative-analysis-for-loops-vs-foreach-loops-in-javascript-fdb4f853dfaa>

<https://www.freecodecamp.org/news/javascript-closures-explained-with-example/>

### <https://www.codecademy.com/learn/game-dev-learn-javascript-higher-order-functions-and-iterators/modules/game-dev-learn-javascript-iterators/cheatsheet>

<https://javascript.info/async>

Callbacks, Callback hell, Promise, Async Await All ES6 Concepts Closure Lexical Scope Var, let, const Hoisting, Scope chain, Higher order functions, call, bind, apply

Promises:

const promise1 = Promise.resolve(3);

const promise2 = 42;

const promise3 = new Promise((resolve, reject) => {

setTimeout(resolve, 100, 'foo');

});

Promise.all([promise1, promise2, promise3]).then((values) => {

console.log(values);

});

// Expected output: Array [3, 42, "foo"]

const promise1 = Promise.reject(0);

const promise2 = new Promise((resolve) => setTimeout(resolve, 100, 'quick'));

const promise3 = new Promise((resolve) => setTimeout(resolve, 500, 'slow'));

const promises = [promise1, promise2, promise3];

Promise.any(promises).then((value) => console.log(value));

// Expected output: "quick"

**Return all capital cases from String**

str.match(/[A-Z]/g)

function isPalindrome(str) {

str = str.replace(/\W/g, '').toLowerCase();

return (str == str.split('').reverse().join(''));

}

console.log(isPalindrome("level")); // logs 'true'

console.log(isPalindrome("levels")); // logs 'false'

console.log(isPalindrome("A car, a man, a maraca")); // logs 'true'

function sum(x) {

if (arguments.length == 2) {

return arguments[0] + arguments[1];

} else {

return function(y) { return x + y; };

}

}

console.log(sum(2,3)); // Outputs 5

console.log(sum(2)(3)); // Outputs 5

for (let i = 0; i < 5; i++) {

var btn = document.createElement('button');

btn.appendChild(document.createTextNode('Button ' + i));

btn.addEventListener('click', function(){ console.log(i); });

document.body.appendChild(btn);

}

console.log(1 + "2" + "2");

console.log(1 + +"2" + "2");

console.log(1 + -"1" + "2");

console.log(+"1" + "1" + "2");

console.log( "A" - "B" + "2");

console.log( "A" - "B" + 2);

"122"

"32"

"02"

"112"

"NaN2"

NaN

var globalVar = "xyz";

(function outerFunc(outerArg) {

var outerVar = 'a';

(function innerFunc(innerArg) {

var innerVar = 'b';

console.log(

"outerArg = " + outerArg + "\n" +

"innerArg = " + innerArg + "\n" +

"outerVar = " + outerVar + "\n" +

"innerVar = " + innerVar + "\n" +

"globalVar = " + globalVar);

})(456);

})(123);

outerArg = 123

innerArg = 456

outerVar = a

innerVar = b

globalVar = xyz

console.log("0 || 1 = "+(0 || 1));

console.log("1 || 2 = "+(1 || 2));

console.log("0 && 1 = "+(0 && 1));

console.log("1 && 2 = "+(1 && 2));

0 || 1 = 1

1 || 2 = 1

0 && 1 = 0

1 && 2 = 2

console.log((function f(n){return ((n > 1) ? n \* f(n-1) : n)})(10));

console.log(1 < 2 < 3); //true

console.log(3 > 2 > 1); //false

for (let i = 0; i < 5; i++) {

setTimeout(function() { console.log(i); }, i \* 1000 );

}

0,1,2,3,4

for (var i = 0; i < 5; i++) {

setTimeout(function() { console.log(i); }, i \* 1000 );

}

5,5,5,5,5

for (var i = 0; i < 5; i++) {

(function(x) {

setTimeout(function() { console.log(x); }, x \* 1000 );

})(i);

}

0,1,2,3,4

for (let i = 0; i < 5; i++) {

setTimeout(function() { console.log(i); }, i \* 1000 );

}

0,1,2,3,4

a

Managing Objects

// Create object with an existing object as prototype  
Object.create(parent, donor)  
  
// Adding or changing an object property  
Object.defineProperty(object, property, descriptor)  
  
// Adding or changing object properties  
Object.defineProperties(object, descriptors)  
  
// Accessing Properties  
Object.getOwnPropertyDescriptor(object, property)  
  
// Returns all properties as an array  
Object.getOwnPropertyNames(object)  
  
// Accessing the prototype  
Object.getPrototypeOf(object)  
  
// Returns enumerable properties as an array  
Object.keys(object)

// Prevents adding properties to an object  
Object.preventExtensions(object)  
  
// Returns true if properties can be added to an object  
Object.isExtensible(object)  
  
// Prevents changes of object properties (not values)  
Object.seal(object)  
  
// Returns true if object is sealed  
Object.isSealed(object)  
  
// Prevents any changes to an object  
Object.freeze(object)  
  
// Returns true if object is frozen  
Object.isFrozen(object)

// ES5  
var x = function(x, y) {  
   return x \* y;  
}

// ES6  
const x = (x, y) => x \* y;

## **JavaScript Promises**

A Promise is a JavaScript object that links "Producing Code" and "Consuming Code".

"Producing Code" can take some time and "Consuming Code" must wait for the result.

### Promise Syntax

const myPromise = new Promise(function(myResolve, myReject) {  
// "Producing Code" (May take some time)  
  
  myResolve(); // when successful  
  myReject();  // when error  
});  
  
// "Consuming Code" (Must wait for a fulfilled Promise).  
myPromise.then(  
  function(value) { /\* code if successful \*/ },  
  function(error) { /\* code if some error \*/ }

const myPromise = new Promise(function(myResolve, myReject) {  
  setTimeout(function() { myResolve("I love You !!"); }, 3000);  
});  
  
myPromise.then(function(value) {  
  document.getElementById("demo").innerHTML = value;  
});

The rest parameter (...) allows a function to treat an indefinite number of arguments as an array:

const myArr = [

{name:"X00",price:100 },

{name:"X01",price:100 },

{name:"X02",price:100 },

{name:"X03",price:100 },

{name:"X04",price:110 },

{name:"X05",price:110 },

{name:"X06",price:110 },

{name:"X07",price:110 },

{name:"X08",price:120 },

{name:"X09",price:120 },

{name:"X10",price:120 },

{name:"X11",price:120 },

{name:"X12",price:130 },

{name:"X13",price:130 },

{name:"X14",price:130 },

{name:"X15",price:130 },

{name:"X16",price:140 },

{name:"X17",price:140 },

{name:"X18",price:140 },

{name:"X19",price:140 }

];

myArr.sort( (p1, p2) => {

if (p1.price < p2.price) return -1;

if (p1.price > p2.price) return 1;

return 0;

});

## **The Nullish Coalescing Operator (??)**

The ?? operator returns the first argument if it is not**nullish**(null or undefined).

Otherwise it returns the second.

### Example

let name = null;  
let text = "missing";  
let result = name ?? text;

## **The Optional Chaining Operator (?.)**

The**Optional Chaining Operator**returns undefined if an object is undefined or null (instead of throwing an error).

### Example

const car = {type:"Fiat", model:"500", color:"white"};  
let name = car?.name;

## **The &&= Operator**

The**Logical AND Assignment Operator**is used between two values.

If the first value is true, the second value is assigned.

### Logical AND Assignment Example

let x = 10;  
x &&= 5;

## **The ||= Operator**

The**Logical OR Assignment Operator**is used between two values.

If the first value is false, the second value is assigned.

### Logical OR Assignment Example

let x = 10;  
x ||= 5;

## **The ??= Operator**

The**Nullish Coalescing Assignment Operator**is used between two values.

If the first value is undefined or null, the second value is assigned.

### Nullish Coalescing Assignment Example

let x = 10;  
x ??= 5;

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_assign_nullish)

// Define object  
const obj = {counter : 0};  
  
// Define setters and getters  
Object.defineProperty(obj, "reset", {  
  get : function () {this.counter = 0;}  
});  
Object.defineProperty(obj, "increment", {  
  get : function () {this.counter++;}  
});  
Object.defineProperty(obj, "decrement", {  
  get : function () {this.counter--;}  
});  
Object.defineProperty(obj, "add", {  
  set : function (value) {this.counter += value;}  
});  
Object.defineProperty(obj, "subtract", {  
  set : function (value) {this.counter -= value;}  
});  
  
// Play with the counter:  
obj.reset;  
obj.add = 5;  
obj.subtract = 1;  
obj.increment;  
obj.decrement;

(function () {  
  let x = "Hello!!";  // I will invoke myself  
})();

Create method is used to create object instance with an already declared object properties and its prototype and assign it to a newly created prototype object and return’s empty object.

const obj = {  
 name: "angularfeed",  
 version: 1  
 }obj.prototype = {  
 type: "blog"  
 }const angObj = Object.create(obj);console.log(angObj); // {}  
console.log(angObj.name); // angularfeed  
console.log(angObj.prototype.type); // blog

Assign method is used to assign object properties from source object to the target object and also return’s the new object.

const sourceObj = {  
 name: "ngfeed",  
 ver: 1  
 }

const targetObj = {  
 name: "jsfeed",  
 ver: 1,  
 type: "academy"  
 }

sourceObj.prototype = {  
 location: "Bangalore"  
 }

const resultObj = Object.assign(targetObj, sourceObj);

console.log(sourceObj); // {name: "ngfeed", ver: 1}  
console.log(targetObj); //{name: "ngfeed", ver: 1,type: "academy"}  
console.log(resultObj); // {name: "ngfeed", ver: 1,type: "academy"}  
console.log(targetObj.prototype.location); // Bangalore

## **The JavaScript call() Method**

The call() method is a predefined JavaScript method.

It can be used to invoke (call) a method with an owner object as an argument (parameter).

With call(), an object can use a method belonging to another object.

This example calls the**fullName**method of person, using it on**person1**:

const person = {  
  fullName: function(city, country) {  
    return this.firstName + " " + this.lastName + "," + city + "," + country;  
  }  
}  
  
const person1 = {  
  firstName:"John",  
  lastName: "Doe"  
}  
  
person.fullName.call(person1, "Oslo", "Norway");

With the apply() method, you can write a method that can be used on different objects.

With the bind() method, an object can borrow a method from another object.

const person = {  
  firstName:"John",  
  lastName: "Doe",  
  display: function () {  
    let x = document.getElementById("demo");  
    x.innerHTML = this.firstName + " " + this.lastName;  
  }  
}  
setTimeout(person.display, 3000);

const person = {  
  firstName:"John",  
  lastName: "Doe",  
  display: function () {  
    let x = document.getElementById("demo");  
    x.innerHTML = this.firstName + " " + this.lastName;  
  }  
}  
  
let display = person.display.bind(person);  
setTimeout(display, 3000);

function add() {  
  let counter = 0;  
  function plus() {counter += 1;}  
  plus();     
  return counter;  
}

const add = (function () {  
  let counter = 0;  
  return function () {counter += 1; return counter}  
})();  
  
add();  
add();  
add();

slice() extracts a part of a string and returns the extracted part in a new string.

The method takes 2 parameters: start position, and end position (end not included).

let text = "Apple, Banana, Kiwi";  
let part = text.slice(7, 13); // Banana

substring() is similar to slice().

The difference is that start and end values less than 0 are treated as 0 in substring().

let str = "Apple, Banana, Kiwi";  
let part = str.substring(7, 13); // Banana

substr() is similar to slice().

The difference is that the second parameter specifies the**length**of the extracted part.

let str = "Apple, Banana, Kiwi";  
let part = str.substr(7, 6); //Banana

## **JavaScript Array splice()**

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

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.splice(2, 0, "Lemon", "Kiwi");

Banana,Orange,Lemon,Kiwi,Apple,Mango

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

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.splice(0, 1);

Orange,Apple,Mango

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

const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];  
const citrus = fruits.slice(1);

Orange,Lemon,Apple,Mango

const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];  
const citrus = fruits.slice(1, 3);

Orange,Lemon

const points = [40, 100, 1, 5, 25, 10];  
points.sort(function(a, b){return a - b});

const points = [40, 100, 1, 5, 25, 10];  
points.sort(function(a, b){return b - a});

The call() method calls a function with a given this value and arguments provided individually.

function personIntro() {

console.log(`${this.firstName} ${this.lastName}`);

};

const person1 = {

firstName: 'Sanjeev',

lastName: 'Sharma'

};

personIntro(); // Output 1: undefined undefined

personIntro.call(person1); // Output 2: Sanjeev Sharma

personIntro.call({ firstName : 'Harry', lastName : 'Potter' });

var car = { registrationNumber: "GA12345", brand: "Toyota", displayDetails: function(){ console.log(this.registrationNumber + " " + this.brand); } }

car.displayDetails(); // GA12345 Toyota

var myCarDetails = car.displayDetails; myCarDetails();

var myCarDetails = car.displayDetails.bind(car); myCarDetails(); // GA12345 Toyota

**The map() method is used to transform the elements of an array, whereas the forEach() method is used to loop through the elements of an array**.

### Functions Assigned to Variables

In JavaScript, functions are a data type just as strings, numbers, and arrays are data types. Therefore, functions can be assigned as values to variables, but are different from all other data types because they can be invoked.

let plusFive = (number) => { return number + 5; };

// f is assigned the value of plusFive

let f = plusFive;

plusFive(3); // 8

// Since f has a function value, it can be invoked.

f(9); // 14

### Callback Functions

In JavaScript, a callback function is a function that is passed into another function as an argument. This function can then be invoked during the execution of that higher order function (that it is an argument of).

Since, in JavaScript, functions are objects, functions can be passed as arguments.

const isEven = (n) => {

return n % 2 == 0;

}

let printMsg = (evenFunc, num) => {

const isNumEven = evenFunc(num);

console.log(`The number ${num} is an even number: ${isNumEven}.`)

}

// Pass in isEven as the callback function

printMsg(isEven, 4);

// Prints: The number 4 is an even number: True.

### Higher-Order Functions

In Javascript, functions can be assigned to variables in the same way that strings or arrays can. They can be passed into other functions as parameters or returned from them as well.

A “higher-order function” is a function that accepts functions as parameters and/or returns a function.

### JavaScript Functions: First-Class Objects

JavaScript functions are first-class objects. Therefore:

* They have built-in properties and methods, such as the name property and the .toString() method.
* Properties and methods can be added to them.
* They can be passed as arguments and returned from other functions.
* They can be assigned to variables, array elements, and other objects.

//Assign a function to a variable originalFunc

const originalFunc = (num) => { return num + 2 };

//Re-assign the function to a new variable newFunc

const newFunc = originalFunc;

//Access the function's name property

newFunc.name; //'originalFunc'

//Return the function's body as a string

newFunc.toString(); //'(num) => { return num + 2 }'

//Add our own isMathFunction property to the function

newFunc.isMathFunction = true;

//Pass the function as an argument

const functionNameLength = (func) => { return func.name.length };

functionNameLength(originalFunc); //12

//Return the function

const returnFunc = () => { return newFunc };

returnFunc(); //[Function: originalFunc]

### The .reduce() Method

The .reduce() method iterates through an array and returns a single value.

In the above code example, the .reduce() method will sum up all the elements of the array. It takes a callback function with two parameters (accumulator, currentValue) as arguments. On each iteration, accumulator is the value returned by the last iteration, and the currentValue is the current element. Optionally, a second argument can be passed which acts as the initial value of the accumulator.

const arrayOfNumbers = [1, 2, 3, 4];

const sum = arrayOfNumbers.reduce((accumulator, currentValue) => {

return accumulator + currentValue;

});

console.log(sum); // 10

### The .forEach() Method

The .forEach() method executes a callback function on each of the elements in an array in order.

In the above example code, the callback function containing a console.log() method will be executed 5 times, once for each element.

const numbers = [28, 77, 45, 99, 27];

numbers.forEach(number => {

console.log(number);

});

### The .filter() Method

The .filter() method executes a callback function on each element in an array. The callback function for each of the elements must return either true or false. The returned array is a new array with any elements for which the callback function returns true.

In the above code example, the array filteredArray will contain all the elements of randomNumbers but 4.

const randomNumbers = [4, 11, 42, 14, 39];

const filteredArray = randomNumbers.filter(n => {

return n > 5;

});

### The .map() Method

The .map() method executes a callback function on each element in an array. It returns a new array made up of the return values from the callback function.

The original array does not get altered, and the returned array may contain different elements than the original array.

In the example code above, the .map() method is used to add ' joined the contest.' string at the end of each element in the finalParticipants array.

const finalParticipants = ['Taylor', 'Donald', 'Don', 'Natasha', 'Bobby'];

// add string after each final participant

const announcements = finalParticipants.map(member => {

return member + ' joined the contest.';

})

console.log(announcements);

**Every object in JavaScript has a built-in property, which is called its prototype**. The prototype is itself an object, so the prototype will have its own prototype, making what's called a prototype chain.

A closure is an inner function that has access to the variables in the outer (enclosing) function’s scope chain. The closure has access to variables in three scopes; specifically: (1) variable in its own scope, (2) variables in the enclosing function’s scope, and (3) global variables.

Here is an example:

var globalVar = "xyz";

(function outerFunc(outerArg) {

var outerVar = 'a';

(function innerFunc(innerArg) {

var innerVar = 'b';

console.log(

"outerArg = " + outerArg + "\n" +

"innerArg = " + innerArg + "\n" +

"outerVar = " + outerVar + "\n" +

"innerVar = " + innerVar + "\n" +

"globalVar = " + globalVar);

})(456);

})(123);

outerArg = 123

innerArg = 456

outerVar = a

innerVar = b

globalVar = xyz

async function f() {

return Promise.resolve(1);

}

f().then(alert); // 1

# JavaScript flatten an array

**let** arr = [

[1, 2],

[3, 4],

[5, 6][7, 8, 9],

[10, 11, 12, 13, 14, 15]

];

**let** flatArray = [].concat.**apply**([], arr);

//Output: [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 ]

**let** flatArray = [].**concat**(...arr);

//Output: [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 ]

console.**log**(arr1.**flat**(2));

function flatten(ary) {

let result = [];

for(let i = 0; i < ary.length; i++) {

if(Array.isArray(ary[i])) {

result = result.concat(flatten(ary[i]));

} else {

result.push(ary[i]);

}

}

return result;

}

console.log(flatten([1,2,[3,4],[[5,6]],7,8]));

**Node JS:**

How Node JS works, , Need of express js , Middleware , Routing, JWT, Protected Route, Error Handling, Writing Test Cases in Node js, Typescript Security, Multithreading ( exec, fork, spawn), Microservices

Server listen

Why Express

<https://www.geeksforgeeks.org/express-js/>

Difference between Eenet loops in browser and node<https://dev.to/jasmin/difference-between-the-event-loop-in-browser-and-node-js-1113>

<https://wearecommunity.io/communities/aep-js-community/articles/2893>

<https://adevait.com/nodejs/introduction-to-queues-nodejs>

Settimeoout / set immediate output

<https://dev.to/ynmanware/setimmediate-settimeout-and-process-nexttick-3mfd>

Node / Javascript Arct.

<https://medium.com/technofunnel/node-js-single-threaded-event-based-architecture-9f73daee37a1>

Rest API / Soap API

<https://aws.amazon.com/compare/the-difference-between-soap-rest/#:~:text=SOAP%20and%20REST%20are%20two%20different%20approaches%20to%20API%20design,exchange%20data%20in%20multiple%20formats>.

Rest API methods

<https://www.digitalocean.com/community/tutorials/nodejs-jwt-expressjs>

<https://www.rithmschool.com/courses/intermediate-node-express/api-tests-with-jest>

<https://dev.to/nedsoft/testing-nodejs-express-api-with-jest-and-supertest-1km6>

<https://nodejs.dev/en/learn/understanding-processnexttick/>

<https://www.freecodecamp.org/news/nodejs-eventloop-tutorial/>

<https://www.freecodecamp.org/news/node-js-what-when-where-why-how-ab8424886e2/>

<https://fem-node-api.netlify.app/>

<https://www.geeksforgeeks.org/jwt-authentication-with-refresh-tokens/>

|  |
| --- |
| // Requiring the module  const express = require('express');  const app = express();    // Route handling  app.get('/', (req, res) => {  res.send('<h2>Hello from Express.js server!!</h2>');  });    // Server setup  app.listen(8080, () => {  console.log('server listening on port 8080');  }); |

const http = require('http');

// Creating server object

const server = http.createServer((req, res) => {

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

res.write('<html>');

res.write('<head><title>GeeksforGeeks</title><head>');

res.write('<body><h2>Hello from Node.js server!!</h2></body>');

res.write('</html>');

res.end();

});

// Server setup

server.listen(3000, ()=> {

console.log("Server listening on port 3000")

});

Node. js runs JavaScript code in a single thread, which means that**your code can only do one task at a time**. However, Node. js itself is multithreaded and provides hidden threads through the libuv library, which handles I/O operations like reading files from a disk or network requests.

**Routing**refers to how an application’s endpoints (URIs) respond to client requests.

***Middleware***functions are functions that have access to the[request object](https://expressjs.com/en/4x/api.html#req)(req), the[response object](https://expressjs.com/en/4x/api.html#res)(res), and the next function in the application’s request-response cycle. The next function is a function in the Express router which, when invoked, executes the middleware succeeding the current middleware.

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 in the stack.

If the current middleware function does not end the request-response cycle, it must call next() to pass control to the next middleware function. Otherwise, the request will be left hanging.

An Express application can use the following types of middleware:

* [Application-level middleware](https://expressjs.com/en/guide/using-middleware.html#middleware.application)
* const express = require('express')
* const app = express()
* app.use((req, res, next) => {
* console.log('Time:', Date.now())
* next()
* })
* [Router-level middleware](https://expressjs.com/en/guide/using-middleware.html#middleware.router)
* middleware:
* const express = require('express')
* const app = express()
* const router = express.Router()
* // a middleware function with no mount path. This code is executed for every request to the router
* router.use((req, res, next) => {
* console.log('Time:', Date.now())
* next()
* })
* // a middleware sub-stack shows request info for any type of HTTP request to the /user/:id path
* router.use('/user/:id', (req, res, next) => {
* console.log('Request URL:', req.originalUrl)
* next()
* }, (req, res, next) => {
* console.log('Request Type:', req.method)
* next()
* })
* // a middleware sub-stack that handles GET requests to the /user/:id path
* router.get('/user/:id', (req, res, next) => {
* // if the user ID is 0, skip to the next router
* if (req.params.id === '0') next('route')
* // otherwise pass control to the next middleware function in this stack
* else next()
* }, (req, res, next) => {
* // render a regular page
* res.render('regular')
* })
* // handler for the /user/:id path, which renders a special page
* router.get('/user/:id', (req, res, next) => {
* console.log(req.params.id)
* res.render('special')
* })
* // mount the router on the app
* app.use('/', router)
* [Error-handling middleware](https://expressjs.com/en/guide/using-middleware.html#middleware.error-handling)
* app.use((err, req, res, next) => {
* console.error(err.stack)
* res.status(500).send('Something broke!')
* })
* [Built-in middleware](https://expressjs.com/en/guide/using-middleware.html#middleware.built-in)

Express has the following built-in middleware functions:

* [express.static](https://expressjs.com/en/4x/api.html#express.static)serves static assets such as HTML files, images, and so on.
* [express.json](https://expressjs.com/en/4x/api.html#express.json)parses incoming requests with JSON payloads.**NOTE: Available with Express 4.16.0+**
* [express.urlencoded](https://expressjs.com/en/4x/api.html#express.urlencoded)parses incoming requests with URL-encoded payloads.**NOTE: Available with Express 4.16.0+**
* [Third-party middleware](https://expressjs.com/en/guide/using-middleware.html#middleware.third-party)
* const express = require('express')
* const app = express()
* const cookieParser = require('cookie-parser')
* // load the cookie-parsing middleware
* app.use(cookieParser())

# Using template engines with Express

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

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

res.render('index', { title: 'Hey', message: 'Hello there!' })

})

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

throw new Error('BROKEN') // Express will catch this on its own.

})

const MongoClient = require('mongodb').MongoClient

MongoClient.connect('mongodb://localhost:27017/animals', (err, client) => {

if (err) throw err

const db = client.db('animals')

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

if (err) throw er109r

console.log(result)

})

})

const mysql = require('mysql')

const connection = mysql.createConnection({

host: 'localhost',

user: 'dbuser',

password: 's3kreee7',

database: 'my\_db'

})

connection.connect()

connection.query('SELECT 1 + 1 AS solution', (err, rows, fields) => {

if (err) throw err

console.log('The solution is: ', rows[0].solution)

})

Tedious -> SqlServer

const Connection = require('tedious').Connection

const Request = require('tedious').Request

const config = {

server: 'localhost',

authentication: {

type: 'default',

options: {

userName: 'your\_username', // update me

password: 'your\_password' // update me

}

}

}

const connection = new Connection(config)

connection.on('connect', (err) => {

if (err) {

console.log(err)

} else {

executeStatement()

}

})

function executeStatement () {

request = new Request("select 123, 'hello world'", (err, rowCount) => {

if (err) {

console.log(err)

} else {

console.log(`${rowCount} rows`)

}

connection.close()

})

request.on('row', (columns) => {

columns.forEach((column) => {

if (column.value === null) {

console.log('NULL')

} else {

console.log(column.value)

}

})

})

connection.execSql(request)

}

## What is an ORM Tool?

An ORM tool is software designed to help OOP developers interact with relational databases. So instead of creating your own ORM software from scratch, you can make use of these tools.

### Q26: What is Event Emmitter? ☆☆☆

**Answer:**All objects that emit events are members of EventEmitter class. These objects expose an eventEmitter.on() function that allows one or more functions to be attached to named events emitted by the object.

When the EventEmitter object emits an event, all of the functions attached to that specific event are called synchronously.

const EventEmitter = require('events');

class MyEmitter extends EventEmitter {}

const myEmitter = new MyEmitter();

myEmitter.on('event', () => {

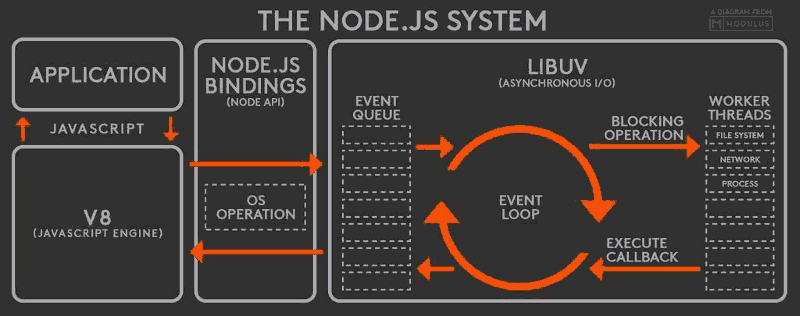
console.log('an event occurred!');

});

myEmitter.emit('event');

**Q33: What's the event loop? ☆☆☆**

**Answer:****The event loop**is what allows Node.js to perform non-blocking I/O operations — despite the fact that JavaScript is single-threaded — by offloading operations to the system kernel whenever possible.

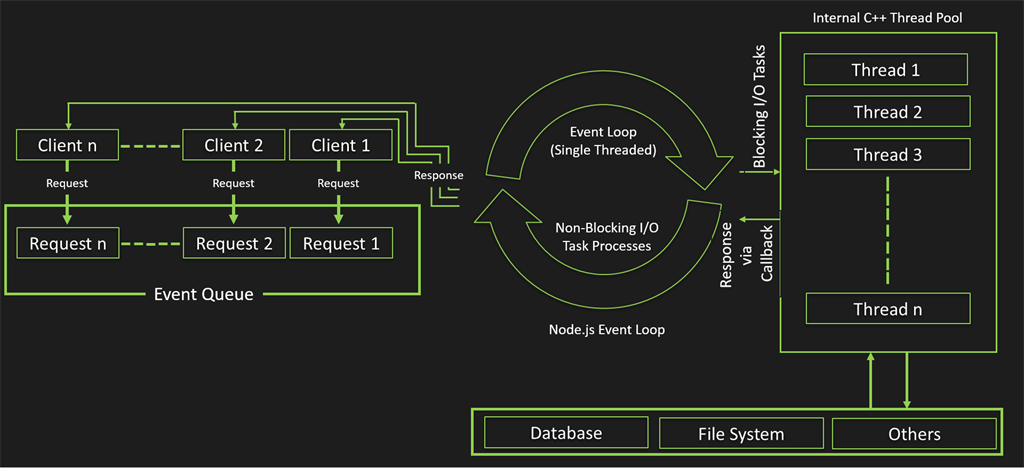
[](https://camo.githubusercontent.com/e2bf4a78840ac7af27c6d94078d315323f87bcdfe4593b473aaf319f037e10c1/68747470733a2f2f692e737461636b2e696d6775722e636f6d2f4c6273397a2e706e67)

Every I/O requires a callback - once they are done, they are pushed onto the event loop for execution. Since most modern kernels are multi-threaded, they can handle multiple operations executing in the background. When one of these operations completes, the kernel tells Node.js so that the appropriate callback may be added to the poll queue to eventually be executed.

**Explain how does Node.js work? ☆☆☆**

**Answer:**A Node.js application creates a single thread on its invocation. Whenever Node.js receives a request, it first completes its processing before moving on to the next request.

Node.js works asynchronously by using the event loop and callback functions, to handle multiple requests coming in parallel. An Event Loop is a functionality which handles and processes all your external events and just converts them to a callback function. It invokes all the event handlers at a proper time. Thus, lots of work is done on the back-end, while processing a single request, so that the new incoming request doesn’t have to wait if the processing is not complete.

[](https://camo.githubusercontent.com/238976a024fbe901c2bfeb3b24a09627b06edc667c6717b951e5fc977bdcb3c3/68747470733a2f2f637368617270636f726e65722d6d696e64637261636b6572696e632e6e6574646e612d73736c2e636f6d2f61727469636c652f6e6f64652d6a732d6576656e742d6c6f6f702f496d616765732f312e706e67)

While processing a request, Node.js attaches a callback function to it and moves it to the back-end. Now, whenever its response is ready, an event is called which triggers the associated callback function to send this response.

### Q18: What is the preferred method of resolving unhandled exceptions in Node.js? ☆☆☆

**Answer:**Unhandled exceptions in Node.js can be caught at the Process level by attaching a handler for uncaughtException event.

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

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

});

However, uncaughtException is a very crude mechanism for exception handling and may be removed from Node.js in the future. An exception that has bubbled all the way up to the Process level means that your application, and Node.js may be in an undefined state, and the only sensible approach would be to restart everything.

The preferred way is to add another layer between your application and the Node.js process which is called the[domain](http://nodejs.org/api/domain.html).

Domains provide a way to handle multiple different I/O operations as a single group. So, by having your application, or part of it, running in a separate domain, you can safely handle exceptions at the domain level, before they reach the Process level.

<https://www.youtube.com/watch?v=7npkRw7gMRA>

[1:12](https://www.youtube.com/watch?v=7npkRw7gMRA&t=72s) What is node js? [2:14](https://www.youtube.com/watch?v=7npkRw7gMRA&t=134s) Difference between Javascript and NodeJS [3:21](https://www.youtube.com/watch?v=7npkRw7gMRA&t=201s) How does NodeJS works? [4:46](https://www.youtube.com/watch?v=7npkRw7gMRA&t=286s) Difference between NodeJS and Angular [6:37](https://www.youtube.com/watch?v=7npkRw7gMRA&t=397s) Why is NodeJS single-threaded? [7:35](https://www.youtube.com/watch?v=7npkRw7gMRA&t=455s) What are the asynchronous operations supported by NodeJS? [8:19](https://www.youtube.com/watch?v=7npkRw7gMRA&t=499s) Difference between synchronous and asynchronous [9:58](https://www.youtube.com/watch?v=7npkRw7gMRA&t=598s) What is the meaning of control flow functions? [10:54](https://www.youtube.com/watch?v=7npkRw7gMRA&t=654s) Why is NodeJS so popular? [11:49](https://www.youtube.com/watch?v=7npkRw7gMRA&t=709s) What is an Event Loop? [13:51](https://www.youtube.com/watch?v=7npkRw7gMRA&t=831s) What are the asynchronous tasks that should occur in an event loop? [14:32](https://www.youtube.com/watch?v=7npkRw7gMRA&t=872s) What is the order of execution in a control flow statement? [15:29](https://www.youtube.com/watch?v=7npkRw7gMRA&t=929s) What are the input arguments in an asynchronous queue? [15:51](https://www.youtube.com/watch?v=7npkRw7gMRA&t=951s) NodeJS disadvantages [17:10](https://www.youtube.com/watch?v=7npkRw7gMRA&t=1030s) What is the reason to use the event-based model in NodeJS? [17:53](https://www.youtube.com/watch?v=7npkRw7gMRA&t=1073s) How to import external libraries into NodeJS? [18:40](https://www.youtube.com/watch?v=7npkRw7gMRA&t=1120s) What does event-driven means? [19:40](https://www.youtube.com/watch?v=7npkRw7gMRA&t=1180s) Difference between Ajax and NodeJS [20:16](https://www.youtube.com/watch?v=7npkRw7gMRA&t=1216s) What is the framework that is used the most in NodeJS? [21:16](https://www.youtube.com/watch?v=7npkRw7gMRA&t=1276s) Security Implementations in NodeJS [22:49](https://www.youtube.com/watch?v=7npkRw7gMRA&t=1369s) What is libuv

**Html and CSS:**

HTML? HTML tags vs elements? List in html? Id vs class? Formatting tags? Head and body? Link? Inline and block elements

Universal selectors / ruleset / css box model / css3 v css2 / z-index / css sprities / media types /

**Cloud:**

Cloud Computing / IAAS,PAAS, SAAS / types of clouding / load balancing / key features /

**SQL:**

CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    PRIMARY KEY (ID)  
);

SELECT \* FROM Customers  
WHERE CustomerName LIKE 'a%';

SELECT column\_name(s)  
FROM table\_name  
WHERE column\_name BETWEEN value1 AND value2;

SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate  
FROM Orders  
INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

CREATE PROCEDURE SelectAllCustomers  
AS  
SELECT \* FROM Customers  
GO;

EXEC SelectAllCustomers;

CREATE TABLE Orders (  
    OrderID int NOT NULL,  
    OrderNumber int NOT NULL,  
    PersonID int,  
    PRIMARY KEY (OrderID),  
    FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)  
);

CREATE INDEX index\_name  
ON table\_name (column1, column2, ...);

SELECT OrderID, Quantity,  
CASE  
    WHEN Quantity > 30 THEN 'The quantity is greater than 30'  
    WHEN Quantity = 30 THEN 'The quantity is 30'  
    ELSE 'The quantity is under 30'  
END AS QuantityText  
FROM OrderDetails;

RDBMS stands for Relational Database Management System.

RDBMS is a program used to maintain a relational database.

A table is a collection of related data entries, and it consists of columns and rows.

A column holds specific information about every record in the table.

A record (or row) is each individual entry that exists in a table.

A trigger in MySQL is a set of SQL statements that reside in a system catalog.**It is a special type of stored procedure that is invoked automatically in response to an event**. Each trigger is associated with a table, which is activated on any DML statement such as**INSERT, UPDATE**, or**DELETE**.

1. [**Before Insert**](https://www.javatpoint.com/mysql-before-insert-trigger)**:**It is activated before the insertion of data into the table.
2. [**After Insert**](https://www.javatpoint.com/mysql-after-insert-trigger)**:**It is activated after the insertion of data into the table.
3. [**Before Update**](https://www.javatpoint.com/mysql-before-update-trigger)**:**It is activated before the update of data in the table.
4. [**After Update**](https://www.javatpoint.com/mysql-after-update-trigger)**:**It is activated after the update of the data in the table.
5. [**Before Delete**](https://www.javatpoint.com/mysql-before-delete-trigger)**:**It is activated before the data is removed from the table.
6. [**After Delete**](https://www.javatpoint.com/mysql-after-delete-trigger)**:**It is activated after the deletion of data from the table.
7. **CREATE****TRIGGER**trigger\_name
8. (**AFTER**| BEFORE) (**INSERT**|**UPDATE**|**DELETE**)
9. **ON**table\_name**FOR**EACH ROW
10. **BEGIN**
11. --variable declarations
12. --trigger code
13. **END**;

A view is a database object that has no values. Its contents are based on the base table. It contains rows and columns similar to the real table. In MySQL, the View is a**virtual table**created by a query by joining one or more tables. It is operated similarly to the base table but does not contain any data of its own.

1. **CREATE**[OR REPLACE]**VIEW**view\_name**AS**
2. **SELECT**columns
3. **FROM**tables
4. [**WHERE**conditions];

1. **ALTER****VIEW**trainer**AS**
2. **SELECT**id, course\_name, trainer
3. **FROM**courses;
4. mysql>**DROP****INDEX**index\_name**ON**table\_name [algorithm\_option | lock\_option];
5. **ALTER****TABLE**table\_name
6. **ADD**new\_column\_name column\_definition
7. [**FIRST**|**AFTER**column\_name ];
8. **ALTER****TABLE**cus\_tbl
9. **MODIFY**cus\_surname**varchar**(50) NULL;

**React:**

Why React

Components life cycles

type and interfaces

Age input box and error if age below 18 years

## **Initialize useState**

We initialize our state by calling useState in our function component.

useState accepts an initial state and returns two values:

* The current state.
* A function that updates the state.
* The useEffect Hook allows you to perform side effects in your components.
* Some examples of side effects are: fetching data, directly updating the DOM, and timers.
* useEffect accepts two arguments. The second argument is optional.
* useEffect(<function>, <dependency>)
* import { useState, useEffect } from "react";
* import ReactDOM from "react-dom/client";
* function Timer() {
* const [count, setCount] = useState(0);
* useEffect(() => {
* setTimeout(() => {
* setCount((count) => count + 1);
* }, 1000);
* });
* return <h1>I've rendered {count} times!</h1>;
* }
* const root = ReactDOM.createRoot(document.getElementById('root'));
* root.render(<Timer />);

## **React Context**

React Context is a way to manage state globally.

It can be used together with the useState Hook to share state between deeply nested components more easily than with useState alone.

## **The Problem**

State should be held by the highest parent component in the stack that requires access to the state.

To illustrate, we have many nested components. The component at the top and bottom of the stack need access to the state.

To do this without Context, we will need to pass the state as "props" through each nested component. This is called "prop drilling".

import { useState, createContext, useContext } from "react";

import ReactDOM from "react-dom/client";

const UserContext = createContext();

function Component1() {

const [user, setUser] = useState("Jesse Hall");

return (

<UserContext.Provider value={user}>

<h1>{`Hello ${user}!`}</h1>

<Component2 />

</UserContext.Provider>

);

}

function Component2() {

return (

<>

<h1>Component 2</h1>

<Component3 />

</>

);

}

function Component3() {

return (

<>

<h1>Component 3</h1>

<Component4 />

</>

);

}

function Component4() {

return (

<>

<h1>Component 4</h1>

<Component5 />

</>

);

}

function Component5() {

const user = useContext(UserContext);

return (

<>

<h1>Component 5</h1>

<h2>{`Hello ${user} again!`}</h2>

</>

);

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Component1 />);

# **React useRef Hook**

The useRef Hook allows you to persist values between renders.

It can be used to store a mutable value that does not cause a re-render when updated.

It can be used to access a DOM element directly.

import { useState, useEffect, useRef } from "react";

import ReactDOM from "react-dom/client";

function App() {

const [inputValue, setInputValue] = useState("");

const count = useRef(0);

useEffect(() => {

count.current = count.current + 1;

});

return (

<>

<input

type="text"

value={inputValue}

onChange={(e) => setInputValue(e.target.value)}

/>

<h1>Render Count: {count.current}</h1>

</>

);

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<App />);

The main difference between rest and spread is that**the rest operator puts the rest of some specific user-supplied values into a JavaScript array.****But the spread syntax expands iterables into individual elements**.15-Sept-2021

// Use rest to enclose the rest of specific user-supplied values into an array:

function myBio(firstName, lastName, ...otherInfo) {

return otherInfo;

}

// Invoke myBio function while passing five arguments to its parameters:

myBio("Oluwatobi", "Sofela", "CodeSweetly", "Web Developer", "Male");

// The invocation above will return:

["CodeSweetly", "Web Developer", "Male"]

// Define a function with three parameters:

function myBio(firstName, lastName, company) {

return `${firstName} ${lastName} runs ${company}`;

}

// Use spread to expand an array’s items into individual arguments:

myBio(...["Oluwatobi", "Sofela", "CodeSweetly"]);

// The invocation above will return:

“Oluwatobi Sofela runs CodeSweetly”

db.comments.aggregate([

{

$lookup: {

from: "movies",

localField: "movie\_id",

foreignField: "\_id",

as: "movie\_details",

},

},

{

$limit: 1

}

])

db.posts.aggregate([

// Stage 1: Only find documents that have more than 1 like

{

$match: { likes: { $gt: 1 } }

},

// Stage 2: Group documents by category and sum each categories likes

{

$group: { \_id: "$category", totalLikes: { $sum: "$likes" } }

}

])

select \* from employee order by sal desc limit 5,1

select salary from table\_name order by salary desc n-1, 1;

select sal from employee e1 where n-1 = (select count(distinct sal) from employee e2 where e2.sal > e1.sal)

db.employee.find({}).sort({“salary”:-1}).skip(1).limit(1);

db.employee.find({salary:{$lt:sal[sal.length-1]}}).sort({“salary”:-1}).limit(1);

**MongoDb:**

ALTER TABLE menus ADD FULLTEXT(item);

SELECT \* FROM menus WHERE MATCH(item) AGAINST("pasta");

<https://www.mongodb.com/basics/full-text-search>

db.menus.aggregate([

{

$search: {

text: {

query: "pasta",

path:"item"

}

}

}

]);

SELECT \*, <output array field>

FROM collection

WHERE <output array field> IN (

SELECT \*

FROM <collection to join>

WHERE <foreignField> = <collection.localField>

);

db.comments.aggregate([

{

$lookup: {

from: "movies",

localField: "movie\_id",

foreignField: "\_id",

as: "movie\_details",

},

},

{

$limit: 1

}

])