**Node theory Question**

1. **Node js made to** => V8 + c++ by Ryan dahl, it’s a run time environment that run JavaScript code into outside from the browser.
2. **Node js workflow**
3. Request
4. Event queue
5. Event loop
   1. Blocking code -> thread pool -> external resources
   2. Non-blocking code -> io pooling
6. Response,

Note: - By default, thread pool size -> 4 and Max = 8 code (dependence to window support)

1. **How do you handle concurrency when node is single thread?**
2. Event loop
3. Callback function
4. **Advantage of using promised instead of callback.**
5. Build error handling.
6. Improved readability
7. Coupling is low.
8. Control flow.
9. **Used Nodejs?**
10. Real time chat
11. Streaming application
12. Microservices architecture
13. Real time collaboration
14. Single page application
15. **Modules in NodeJS. (require ())**
16. http
17. util
18. fs
19. URL
20. query string
21. stream
22. zlib (compress and decompress files)
23. **module. Exports ()**

* it encapsulate all related code in one single unit of code.

1. **Event loop in node js**

* Its handle asynchronous callback function.

1. **Process.nextTick() and setImmediate ()**-> nextTick () postponed the execution of action until the next pass around the event loop.

* setImmediate () executes a callback on the next cycle.

1. **EventEmitter in NodeJS.**

* It’s a class that hold all the object that can emit events.

1. **Stream in NodeJS**

* Readable
* Writable
* Duplex
* Transform

1. **Fork () and spawn () method**
   1. Both methods create a new child process.
   2. Spawn () method creates a new function.
   3. Fork creates a new instance of fork () method and creates multiple workers to perform same task.
2. **Buffer class in NodeJS**
   1. It’s used to perform raw binary data.
   2. It cannot be resizable.
   3. when dealing with TCP streams or the file system, it's necessary to handle octet streams.(utf-8)
3. **Piping in NodeJS**
   1. It’s a mechanism to connect the output of one stream to other.
   2. Its normally used to retrieve data from one stream and pass output to other stream.
4. **Reactor pattern**
   1. It’s a concept of non-blocking I/O operations.
   2. Its provide a handler that is associated with each I/O operations.
   3. As soon as I/O generated , it is then submitted to a demultiplexer.
5. **Test pyramid.**
   1. Unit testing.
   2. Integration testing.
   3. End to end testing.
6. **Middleware in NodeJS.**
   1. It’s a function that receive request response objects.
   2. Execute code.
   3. Update and modify the req and res.
   4. Finish req-res cycle.
   5. Invoke the next middleware in stack.
7. **Http request.**
   1. Get
   2. Post
   3. Delete
   4. Update
   5. Fetch
8. **Punny code**
   1. The Punycode is an encoding syntax which is used to convert Unicode (UTF-8) string of characters to ASCII string of characters.
9. **Body-parser**
   1. Its NodeJS body-parsing middleware.
   2. It’s parsing an incoming request body in a middleware before its handle.
   3. It’s a npm module process in http request.
10. **TLS module**
    1. It provides transport layer security.
    2. SSL (secure socket layer).
    3. It’s used for establishing a secure connection on the network.
11. **Module in node js and how to define?**
    1. Module in NodeJS where you can define function and get these on another module with the help of require function (module.export function for reuse function whenever we want.
12. **What is different between https and http service.**
    1. **http** – unsecure , default port -80, plain text,
    2. **https** – secure (using ssl/tls), data encryption, authentication, data integrity, Seo benefit, browser trust, default port – 443
13. **How does Node.js handle single-threaded operations with asynchronous processing? Why Node.js is Fast?**
    1. Single threaded but highly concurrent using asynchronous, non-blocking I/O.
    2. Using libuv for background processing in a thread pool. (e.g: file system, crypto)
    3. Event loop & Callback Queue – that manage operations efficiently without blocking execution.
14. **What are Streams in Node.js? Can you give examples of different stream types and their use cases?**
    1. Streams in NodeJS are data-handling object that allow reading and writing in continuously in chunk, instead of loading everything into memory at once.
    2. Use eventEmitter to process data asynchronously.
    3. Type: Readable, writable, duplex and Transform.
15. **Explain the difference between setImmediate () and process.nextTick().**
    1. In execution order process.nextTick() is high priority then setImmediate (); but inside an i/o operation setImmediate() executes before process.nextTick() because it’s in the Check phase, whereas process.nextTick() stays in the Microtask queue.
16. **How would you handle a large number of concurrent requests in a Node.js application?**
    1. Use Cluster Mode for Multi-Core Utilization
    2. Use a Reverse Proxy (Nginx/Load Balancer)
    3. Optimize Database Queries.
    4. Implement Asynchronous Code Efficiently.
    5. Use Caching for Faster Responses (Redis)
    6. Use a Message Queue for Heavy Processing. (RabbitMQ, Kafka)
    7. Enable Compression for Faster Responses. (Zlip compression)
17. **How to Optimize DB query.**
    1. Indexing
    2. Connection pooling
    3. Caching
    4. Use ORM/Query Builder
18. **How does the V8 engine optimize JavaScript execution in Node.js?**
    * 1. JIT Compilation (TurboFan + Ignition)
      2. Hidden Classes & Inline Caching
      3. Garbage Collection (Orinoco)
      4. Tail Call Optimization
      5. Loop Unrolling & Hoisting
      6. Optimized Object & Array Handling
19. **What is load balancing**

Load balancing is the process of distributing incoming network traffic or computational tasks across multiple servers, systems, or resources to ensure no single server or resource is overwhelmed. It helps optimize resource utilization, improve system performance, and provide high availability and reliability.

1. **What are Promises, and how do they differ from async/await? Provide an example of both.**
   1. A promise is an object that represents the eventual completion(or failure) of an asynchronous operation. It allows handling asynchronous tasks without **callback hell.**
      * 1. **Promise.all**: - Executes multiple Promises **in parallel** and waits for all of them to complete.
        2. **Promise.race**: - Returns the result of **the first Promise that resolves or rejects**.
        3. **Promise.any:** - Returns the **first fulfilled Promise**, **ignoring rejections**.
        4. **Promise.allSettled**: - Waits for **all Promises** to complete, regardless of success or failure.
   2. async/await is a **modern syntax** for handling asynchronous operations, making code look **synchronous** and more readable.
2. **What is the purpose of the Cluster module in Node.js? How would you implement clustering?**
   1. The **Cluster module** in Node.js **enables multi-core processing**, improving scalability and fault tolerance.
   2. Each worker process handles a portion of incoming requests, **distributing load efficiently**.
3. **How do you secure a Node.js application ?** 
   1. Secure http header : user HELMET

const helmet = require("helmet");

app.use(helmet());

* 1. Data validation and sanitization : joi or express-validator

const Joi = require("joi"); const schema = Joi. Object({ username: Joi.string().alphanum().min(3).max(30).required(), password: Joi.string().min(8).required() }); const { error } = schema.validate(req.body); if (error) return res.status(400).send(error.details);

* 1. Use authentication and authorization.
  2. Secure API routes
     1. Cors
     2. Express-rate-limit
  3. Secure sensitive data :- use dotenv (.env) file
  4. Use Https and secure cookies

1. **Express Js features**
   1. Middleware support
   2. Routing system
   3. Build a 3rd party middleware
   4. Template engine support
   5. Rest API development
   6. Error handling
   7. Database integration
   8. Security feature
   9. WebSocket support
   10. Easy deployment

**API Development**

1. **How would you debug a memory leak in a Node.js application?**
2. **What strategies would you use to log application data effectively in a production Node.js app?**

**Security**

1. **How do you prevent security vulnerabilities like SQL injection and cross-site scripting (XSS) in a Node.js application?**
2. **What are common security best practices for using JWT (JSON Web Tokens) in Node.js?**
3. **Explain how you would secure sensitive data, such as API keys or database credentials, in a Node.js app.**

**Database Integration**

1. **How would you optimize database queries in a Node.js application?**
2. **Explain the differences between Sequelize and Mongoose. Which would you choose for a specific project and why?**
3. **How do you manage database migrations in a Node.js application?**

**Performance and Scaling**

1. **What is load balancing, and how would you implement it in a Node.js app?**
2. **How do you use caching strategies in Node.js to improve performance?**
3. **What are the differences between horizontal and vertical scaling in the context of Node.js?**

**Deployment and DevOps**

1. **What are some tools and practices for deploying Node.js applications in production?**
2. **Explain how Docker can be used with Node.js for application deployment.**
3. **How would you implement CI/CD for a Node.js project?**

**Advanced Patterns**

1. **What are design patterns commonly used in Node.js applications?**
   1. Singleton, Factory, Observer, etc.
2. **How would you handle a situation where you need to execute multiple async operations in parallel and wait for all of them to complete?**
3. **What is backpressure in Node.js, and how would you handle it?**

**Scenario-Based Questions**

1. **You need to build a high-performance API for a real-time chat application. How would you approach it using Node.js?**
2. **A Node.js application you built is crashing intermittently in production. How would you debug and resolve this issue?**
3. **How would you handle file uploads in Node.js, ensuring security and scalability?**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | |  |  |  | | --- | --- | --- | | Feature | NestJS | Express.js | | Core | **Built on top of Express.js (or Fastify)** | |  | | --- | | **Minimalist, unopinionated framework** |  |  | | --- | |  | | | Architecture | |  | | --- | | **Modular, follows MVC & Dependency Injection (DI)** |  |  | | --- | |  | | |  | | --- | | **Flexible, middleware-based** |  |  | | --- | |  | | | |  | | --- | | **TypeScript Support** |  |  | | --- | |  | | |  | | --- | | **Built-in TypeScript support** |  |  | | --- | |  | | |  | | --- | | **Requires manual TypeScript setup** |  |  | | --- | |  | | | Performance | |  | | --- | | **Uses Fastify (optional) for better performance** |  |  | | --- | |  | | |  | | --- | | **Slower than Fastify but sufficient for most apps** |  |  | | --- | |  | | | Scalability | |  | | --- | | **Highly scalable due to modular design** |  |  | | --- | |  | | |  | | --- | | **Requires manual structuring for scalability** |  |  | | --- | |  | | | |  | | --- | | **Dependency Injection** |  |  | | --- | |  | | **Full supports DI**   |  | | --- | |  | | |  | | --- | | **No built-in DI, requires manual implementation** |  |  | | --- | |  | | | Routing | |  | | --- | | **Uses decorators (@Get(), @Post())** |  |  | | --- | |  | | |  | | --- | | **Uses middleware (app.get(), app.post())** |  |  | | --- | |  | | | |  | | --- | | **Middleware & Interceptors** |  |  | | --- | |  | | |  | | --- | | **Guards, Pipes, Filters, and Interceptors** |  |  | | --- | |  | | |  | | --- | | **Uses middleware functions** |  |  | | --- | |  | | | |  | | --- | | **WebSocket’s & GraphQL** |  |  | | --- | |  | | |  | | --- | | **Built-in support via modules** |  |  | | --- | |  | | |  | | --- | | **Requires additional setup** |  |  | | --- | |  | | | Testing | |  | | --- | | **Provides built-in testing utilities** |  |  | | --- | |  | | |  | | --- | | **Testing setup is manual** |  |  | | --- | |  | | | |  | | --- | | **Microservices Support** |  |  | | --- | |  | | |  | | --- | | **Built-in support for gRPC, Kafka, MQTT, etc.** |  |  | | --- | |  | | |  | | --- | | **Requires external libraries** |  |  | | --- | |  | | | |  | | --- | | **Community & Ecosystem** |  |  | | --- | |  | | |  | | --- | | **Growing community, enterprise-level use** |  |  | | --- | |  | | **Large community, widely adopted** | |

11. What are the main disadvantages of Node.js?

12. What is REPL in Node.js?

21. Explain crypto module in Node.js.

23. Explain the use of the timers module in Node.js.

27. Explain the use of the passport module in Node.js.

29. What are the three methods to avoid callback hell?

31. What is CORS in Node.js?

34. How to manage sessions in Node.js?

37. Explain the packages used for file uploading in nodejs.->fs, multer

38. How to handle database connections in nodejs.-> moongoose

39. How to read command line arguments in nodejs.

40. What are child processing in nodejs?

**JAVASCRIPT**

1. **What are the different data types in JavaScript?**

JavaScript has 8 data types:

Primitive Types:

1. string
2. number
3. bigint
4. Boolean
5. undefined
6. symbol
7. null

Non-Primitive Type:

object (includes arrays, functions, dates, etc.)

1. **Explain the difference between null, undefined, and NaN.**
   1. **Null** => A value that represents "nothing" (explicitly assigned).
   2. **Undefined** => A variable that has been declared but not assigned any value.
   3. **NaN = >** "Not-a-Number"; usually occurs due to invalid mathematical operations (e.g., parseInt("abc")).
2. **Explain how typeof and instanceof work.**

typeof checks data type.

console.log (typeof "hello"); // "string"

console.log (typeof 42); // "number"

instanceof checks if an object belongs to a class.

console.log ([] instanceof Array); // true

console.log ({} instanceof Object); // true

1. **What is hoisting in JavaScript?**

Hoisting allows using functions and variables before declaration.

* 1. console.log(x); // undefined

var x = 10;

* 1. greet(); // "Hello"

function greet() { console.log("Hello"); }

👉 Only var and function declarations are hoisted.

1. **What are closures in JavaScript?**

A closure is when an inner function remembers the variables of its outer function.

function outer() {

let count = 0;

return function inner() {

count++;

console.log(count);

};

}

const increment = outer();

increment(); // 1

increment(); // 2

1. **Explain prototype in JavaScript**.

Every JavaScript object has a prototype that allows property inheritance.

function Person(name) {

this.name = name;

}

Person.prototype.sayHello = function () {

console.log("Hello " + this.name);

};

const user = new Person("Alice");

user.sayHello(); // "Hello Alice"

1. **Explain function currying in JavaScript.**

Currying breaks down a function into multiple functions that take one argument at a time.

function add(a) {

return function (b) {

return a + b;

};

}

console.log(add(5)(10)); // 15

1. **What is the difference between let, const, and var?**
   1. var Function-scoped ✅ Yes Hoisted (undefined)
   2. let Block-scoped ✅ Yes Hoisted (but not initialized)
   3. const Block-scoped ❌ No Hoisted (but not initialized)
2. **What is the purpose of this keyword?**
   1. Global Scope: this refers to the window object.
   2. Object Method: this refers to the object.
   3. Arrow Functions: No own this, it uses parent scope.
3. **What is an IIFE (Immediately Invoked Function Expression)?**

A function that runs immediately when defined.

1. **Explain higher-order functions with an example.**

A higher-order function is a function that takes another function as a parameter or returns a function.

1. **What is the difference between function declaration and function expression?**
   1. Function Declaration ✅ Yes function hello() {}
   2. Function Expression ❌ No const hello = function() {};
2. **Explain callback functions in JavaScript.**

A callback is a function passed as an argument to another function.

1. **What is memoization, and how does it work?**

Memoization caches function results to avoid redundant calculations.

function memoize(fn) {

let cache = {};

return function (arg) {

if (cache[arg]) return cache[arg];

cache[arg] = fn(arg);

return cache[arg];

};

}

const square = memoize((x) => x \* x);

console.log(square(5)); // 25

console.log(square(5)); // 25 (from cache)

1. **Difference between async/await and Promises?**

async/await makes code easier to read and avoids .then().

async function fetchData() {

const data = await fetch("https://api.example.com");

console.log(data);

}

fetchData();

fetch("https://api.example.com")

.then(response => response.json())

.then(data => console.log(data))

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

1. **What is Promise.all(), Promise.race(), Promise.allSettled()?**

Method Behavior

* 1. Promise.all([p1, p2]) Resolves when all promises succeed (fails if one rejects).
  2. Promise.race([p1, p2]) Resolves first completed promise (whether success or failure).
  3. Promise.allSettled([p1, p2]) Returns results of all promises, regardless of failure.
  4. Promise.any([p1,p2]) Return the resolve promise(even the any of then failed)

const p1 = new Promise((res) => setTimeout(res, 1000, "P1 Done"));

const p2 = new Promise((res, rej) => setTimeout(rej, 500, "P2 Error"));

Promise.all([p1, p2]).catch(console.log); // Fails because p2 rejects

Promise.race([p1, p2]).then(console.log); // "P2 Error" (first settled)

1. **What is event loop in JavaScript**

The event loop allows JavaScript to perform non-blocking I/O operations.

It continuously checks the Call Stack and moves completed async tasks from the Task Queue.

Synchronous tasks run first (Call Stack).

Promises & setTimeout move to the Task Queue or Microtask Queue.

The Event Loop moves them back to the Call Stack when it's empty.

1. **How would you optimize a web page's performance for handling a large number of DOM elements dynamically generated in real time?**

| **Optimization** | **Benefit** |
| --- | --- |
| **Virtual DOM** | Reduces direct DOM manipulation |
| **Document Fragments** | Minimizes reflows & repaints |
| **Lazy Loading & Infinite Scroll** | Loads only visible elements |
| **Debounce & Throttle** | Reduces redundant computations |
| **Virtualization** | Renders only necessary elements |
| **Web Workers** | Moves heavy computation off the main thread |
| **CSS Optimization** | Reduces rendering bottlenecks |

1. **What is setImmediate() vs process.nextTick()?**
   1. setImmediate() -> Executes after the current event loop.
   2. process.nextTick() -> Executes before the next event loop starts (higher priority).

setImmediate(() => console.log("setImmediate"));

process.nextTick(() => console.log("nextTick"));

// Output: nextTick → setImmediate

1. **Describe common causes of memory leaks in JavaScript and how you can prevent them.**

**Common Causes:**

* Unused global variables or closures.
* Forgotten timers or intervals.
* Unreferenced DOM elements still in memory.
* Circular references.

**Prevention Techniques:**

* Use let and const instead of var.
* Clear timers using clearInterval or clearTimeout.
* Remove event listeners when not needed.
* Use tools like Chrome DevTools to detect leaks.

1. **Swallow Copy and Deep copy**

* **Swallow copy: -** its copy the array of list but not the object contains it. If any changes in object from original array it will affect the copy array and vice versa.
  + Array.from()
  + [… arr] (Destructuring)
  + Loop (pushed method to pushed array in new array)
* **Deep Copy: -** its copy the array and array of object (nested array of object) but not in the function inside it. (E.g: - If original array has function and want to deep copy, it’s not copied the function but copy of all array and nested array)

**let** deepCopy = JSON.parse(JSON.stringify(original))

1. **Sparse array and Dense array.**

* **sparse array:** it’s had gaps between elements. (E.g: - let a = [1,2,,4,,6])
* **dense array: -** it’s had all elements in the array**.** (E.g:- let a = [1,2,3,4,5,6])

1. **What is event delegation?**

Attaching a single event listener to a parent element to handle multiple child elements.

document.getElementById("parent").addEventListener("click", (e) => {

if (e.target.classList.contains("child")) {

console.log("Child clicked!");

}

});

1. **What is the difference between Call, Apply, and Bind?**

 call(): Calls function with this and arguments as comma-separated values.

 apply(): Calls function with this and arguments as an array.

 bind(): Returns a new function with a bound this, which can be executed later.

const person = { name: "Pintoo" };

function sayHello(age) {

console.log(`Hello, my name is ${this.name} and I'm ${age} years old`);

}

sayHello.call(person, 25);

sayHello.apply(person, [25]);

const newFunc = sayHello.bind(person, 25);

newFunc();

1. **Converting JSON into an Array?** => Object.entries()

const user = { name: "Pintoo", age: 25, city: "Hazaribagh" };   
console.log(Object.entries(user));

1. **How do you compare Object and Map ?**

Both **Objects** and **Maps** store key-value pairs, but they have different use cases and performance characteristics. Let's compare them in detail.

**Object({})** : Objects are a fundamental data structure in JavaScript used to store keyed collections of values.

const obj = { name: "Pintoo", age: 25 };

console.log(obj.name); // Pintoo

**Map(new Map()) : -**

Map is a built-in JavaScript object designed for efficient key-value storage.

const map = new Map();

map.set("name", "Pintoo");

map.set("age", 25);

console.log(map.get("name")); // Pintoo

1. **What are lambda expressions or arrow functions?**

In JavaScript, **arrow functions (=>)**, also called **lambda expressions**, are a concise way to write functions. They were introduced in **ES6 (ECMAScript 2015)** and provide a cleaner syntax compared to traditional function expressions.

1. **What is a first-class function?**

**A function in JavaScript is first-class because it can be:**

* **Assigned to a variable**
* **Passed as an argument**
* **Returned from another function**

1. **What is a unary function?**

A **unary function** is a function that takes exactly **one argument**.

1. **What is a pure function & impure function?**
   1. If the function returns the same output on same input.
   2. If the function returns different output on same input.
2. **What is the Temporal Dead Zone?**

If the variable is defined but not used anywhere of the function and method.

1. **What are ES6 classes?**

ES6 introduced **classes** in JavaScript as a more structured and readable way to work with objects and prototypes. They provide a **blueprint** for creating objects and support **inheritance**, making it easier to write and manage object-oriented code.

Private Properties (ES2020+)

With ES2020, **private fields** are denoted with # and are only accessible within the class.

class BankAccount {

#balance;

constructor(balance) {

this.#balance = balance;

}

getBalance() {

return this.#balance;

}

}

const account = new BankAccount(1000);

console.log(account.getBalance()); // 1000

console.log(account.#balance); // ❌ Error: Private field cannot be accessed outside the class.

1. **What are modules, and why do you need them?**

**Modules in JavaScript allow you to split your code into separate files and reuse them across different parts of your application. They help in organizing code, improving maintainability, and avoiding global scope pollution.**

1. **What is a service worker?**

**A Service Worker is a JavaScript script that runs in the background of a web page, separate from the main browser thread. It acts as a proxy between the web page and the network, enabling features like:  
✅ Offline capabilities  
✅ Background sync  
✅ Push notifications  
✅ Caching for faster performance**

1. What is IndexedDB?
2. **What are the differences between cookies, localStorage, and sessionStorage?**

**Feature** **localStorage** **sessionStorage Cookies**

Expiry Never On tab close Custom expiry

Size 5MB 5MB 4KB

Access Only JavaScript Only JavaScript Server & JS

localStorage.setItem("name", "Alice")

console.log (localStorage.getItem("name"))

1. What is strict mode in JavaScript?
2. Why do you need strict mode?
3. What is the purpose of the delete operator?
4. What is the difference between window and document?
5. What is the difference between document load and DOMContentLoaded events?
6. What is event bubbling?
7. What is event capturing?
8. How do you submit a form using JavaScript?
9. What is the same-origin policy?
10. What is the difference between native, host, and user objects?
11. **What are the tools or techniques used for debugging & throttling JavaScript code?**

**Debouncing**: Limits function calls to only after a delay.

function debounce(fn, delay) {

let timer;

return function () {

clearTimeout(timer);

timer = setTimeout(() => fn(), delay);

};

}

**Throttling**: Limits function execution to once in a fixed interval.

function throttle(fn, interval) {

let lastCall = 0;

return function () {

let now = Date.now();

if (now - lastCall > interval) {

lastCall = now;

fn();

}

};

}

1. **Is JavaScript a case-sensitive language?**

* JavaScript treats uppercase and lowercase as different characters.
* Variable names, function names, object properties, and keywords must match exactly.
* Be careful when working with external APIs, database field names, or file paths, as they might be case-sensitive.

1. **What are events in JavaScript? (click, onclick, mouse, key, submit, change, select etc.)**
2. **How does garbage collection work in JavaScript?**

JavaScript uses reference counting & mark-and-sweep algorithms to clean unused memory.

let obj = { name: "Alice" };

obj = null; // Eligible for garbage collection

1. **What is the difference between ES5 and ES6?**

**Feature ES5 ES6**

Variable declaration var let, const

Functions function Arrow functions

Objects prototype class syntax

Modules No module system import/export

1. **How do you create a generator function?**

Generators produce values lazily using yield.

function\* generateNumbers() {

yield 1;

yield 2;

yield 3;

}

const iterator = generateNumbers();

console.log(iterator.next().value); // 1

console.log(iterator.next().value); // 2

1. **What is a WeakMap and WeakSet?**

WeakMap allows garbage collection of keys.

WeakSet only holds objects (not values).

let weakMap = new WeakMap();

let obj = {};

weakMap.set(obj, "data");

obj = null; // Object removed from memory

1. **How does JavaScript handle memory management?**

It uses automatic garbage collection based on object references.

1. **Explain Symbol in JavaScript**.

A unique, immutable data type used for object properties.

const sym = Symbol("id");

const obj = { [sym]: 123 };

1. **What is the purpose of Object.freeze()?**

**Prevents modification of an object. Not update/ delete/ add**

const obj = Object.freeze({ name: "Alice" });

obj.name = "Bob"; // Error

1. **Object.seal()**

**its can give access to modify , but not delete not add/insert**

**Let user : {name : “pintoo”}  
Object.seal(user);**

**user.name = “Ronak”; // accepted   
user.surname = “Rana” // not accepted  
delete user.name // not accepted**

**Programming Question:-  
1. Q: Write a function to flatten a deeply nested array without using Array.prototype.flat.**

**2. Q: How can you remove duplicates from an array? Provide multiple approaches.**

**3. Q: Write a function to find the intersection of two arrays.**

**4. Q: Implement a function that splits an array into chunks of a given size.**

**5. Q: Write a function to rotate an array k times to the right.**

**6. Q: Write a function to find the maximum product of any three numbers in an array.**

**7. Q: Write a function that finds all unique pairs in an array that sum up to a given target.**

**8. Q: Merge two sorted arrays into one sorted array without using built-in sort.**

**9. Q: Find the missing number in an array of n integers ranging from 1 to n+1.**

**10. Q: Write a function to find the longest subarray with unique elements.**

**ANGULAR**

**1. Core Angular Concepts**

🔹 **What are the key differences between AngularJS and Angular?**  
🔹 **Explain the concept of dependency injection in Angular.**  
🔹 **What are Angular lifecycle hooks? Can you explain each one?**  
🔹 **What is the difference between ngOnInit() and constructor()?**  
🔹 **How do you handle change detection in Angular?**  
🔹 **What are Angular modules (NgModule), and why are they important?**

🔹 **What are Standalone Components in Angular 16+? How do they differ from traditional NgModules?**

**2. Components & Directives**

🔹 **What is a component in Angular? How does it differ from a directive?**  
🔹 **Explain the difference between attribute and structural directives.**  
🔹 **How does @Input() and @Output() work in Angular?**  
🔹 **What is ViewChild and ViewChildren in Angular?**  
🔹 **What is ContentChild and ContentChildren, and when should you use them?**  
🔹 **What is a HostListener and HostBinding in Angular?**

**3. Forms in Angular**

🔹 **What is the difference between Template-driven and Reactive forms?**  
🔹 **How do you perform form validation in Angular?**  
🔹 **How do you dynamically create a form in Angular using FormBuilder?**  
🔹 **What is formControlName, and how is it different from ngModel?**  
🔹 **How do you handle conditional form validation?**

🔹 **What are Form Arrays, and when would you use them?**

**4. Angular Services & State Management**

🔹 **What is the role of a service in Angular?**  
🔹 **What is the difference between a service and a factory?**  
🔹 **How do you use RxJS in Angular services?**  
🔹 **What is the difference between Subject, BehaviorSubject, and ReplaySubject?**  
🔹 **Explain NgRx and how it is used for state management in Angular.**

**5. Angular Routing & Navigation**

🔹 **How does Angular Routing work?**  
🔹 **What is the difference between RouterModule.forRoot() and RouterModule.forChild()?**  
🔹 **How do you pass data between components using route parameters?**  
🔹 **What are Route Guards in Angular? Name different types of guards.**  
🔹 **What is Lazy Loading in Angular, and how do you implement it?**  
🔹 **What is PreloadingStrategy in Angular?**

**6. HTTP Client & APIs**

🔹 **How do you make an HTTP call in Angular?**  
🔹 **What is the difference between HttpClient and Http module?**  
🔹 **How do you handle errors in HTTP calls using HTTP Interceptor?**  
🔹 **What is CORS, and how do you handle it in Angular?**  
🔹 **How do you implement caching in Angular with HTTP requests?**

**7. Angular Performance Optimization**

🔹 **How do you improve Angular app performance?**  
🔹 **What is ChangeDetectionStrategy.OnPush, and how does it work?**  
🔹 **How do you optimize Angular rendering using trackBy in \*ngFor?**  
🔹 **What is Ahead-of-Time (AOT) compilation?**  
🔹 **How does Lazy Loading help in Angular performance optimization?**  
🔹 **What is the role of Webpack in Angular?**

**8. Security in Angular**

🔹 **What is Cross-Site Scripting (XSS), and how does Angular prevent it?**  
🔹 **How does Angular prevent SQL injection?**  
🔹 **What is Content Security Policy (CSP) in Angular?**  
🔹 **How do you handle authentication and authorization in Angular?**  
🔹 **What is Angular’s built-in DOM sanitization?**

**8.1. Reactive Programming with RxJS**

🔹 **What is the difference between switchMap, mergeMap, concatMap, and exhaustMap?**  
🔹 **How do you handle error handling in RxJS streams?**🔹 **What are Subjects and BehaviorSubjects? When would you use each?**  
🔹 **Explain how to debounce user input using RxJS.**

🔹 **How do you combine multiple streams using combineLatest or forkJoin?**

**9. Testing in Angular**

🔹 **What are the different types of testing in Angular?**  
🔹 **How do you write unit tests for Angular components?**  
🔹 **How do you mock services and HTTP calls in Angular tests?**  
🔹 **What is Jasmine and Karma in Angular testing?**  
🔹 **What is Protractor, and how does it work in Angular?**

**10. Miscellaneous / Advanced Angular**

🔹 **What are Angular Pipes? Difference between pure and impure pipes?**  
🔹 **What are Angular decorators? Explain different types of decorators.**  
🔹 **What is Angular Universal?**  
🔹 **What is Server-Side Rendering (SSR), and how does Angular support it?**  
🔹 **What is the difference between a template-driven and a reactive form?**  
🔹 **How do you implement Progressive Web Apps (PWA) in Angular?**

**MongoDB**

**1. What is an index in MongoDB?**

db.users.createIndex({ name: 1 }); // Ascending index on 'name'

**2. What are the types of indexes in MongoDB?**

Index Type Description

Single Field Index { name: 1 }

Compound Index { age: 1, name: -1 }

Text Index { bio: "text" }

Hashed Index { \_id: "hashed" }

TTL Index Automatically removes old documents

**3. What is the difference between a Primary and Secondary Index?**

Primary Index: Automatically created on \_id.

Secondary Index: User-defined indexes for faster queries.

db.users.createIndex({ email: 1 }); // Secondary index on 'email'

**4. What is Aggregation in MongoDB?**

* Aggregation processes data in stages (similar to SQL GROUP BY).

Count users by age:

db.users.aggregate([{ $group: { \_id: "$age", count: { $sum: 1 } } }]);

**5. What are the key aggregation stages?**

Stage Description

$match Filters documents (like WHERE)

$group Groups documents (like GROUP BY)

$project Reshapes documents (like SELECT)

$sort Sorts documents

$limit Limits results

$skip Skips N documents

db.sales.aggregate([

{ $match: { status: "Completed" } },

{ $group: { \_id: "$product", totalSales: { $sum: "$amount" } } },

{ $sort: { totalSales: -1 } },

{ $limit: 5 }

]);

**6. What is $lookup in aggregation?**

$lookup performs joins between collections.

Join orders with customers:

db.orders.aggregate([

{

$lookup: {

from: "customers",

localField: "customerId",

foreignField: "\_id",

as: "customerDetails"

}

}

]);

**7. What is a MongoDB transaction?**

* MongoDB supports multi-document ACID transactions in replica sets.

const session = db.getMongo().startSession();

session.startTransaction();

try {

session.getDatabase("test").users.insertOne({ name: "Alice" });

session.commitTransaction();

} catch (e) {

session.abortTransaction();

}

**8. What is replication in MongoDB?**

* Replication ensures high availability by creating multiple copies of data.

Primary Node → Handles writes

Secondary Nodes → Handle reads & backup

rs.initiate({

\_id: "myReplicaSet",

members: [

{ \_id: 0, host: "localhost:27017" },

{ \_id: 1, host: "localhost:27018" },

{ \_id: 2, host: "localhost:27019" }

]

});

**9. What is sharding in MongoDB?**

* Sharding splits large datasets across multiple servers for better performance.

sh.enableSharding("myDatabase");

sh.shardCollection("myDatabase.users", { userId: "hashed" });

* Uses a shard key (e.g., userId)
* Each shard contains a subset of data

**10. How to secure MongoDB?**

* Enable authentication: mongod --auth
* Use roles & privileges:

db.createUser({

user: "admin",

pwd: "password",

roles: [{ role: "readWrite", db: "testDB" }]

});

* Enable TLS/SSL for encryption

**11. get a query for selected and projected data.**

userModel.find().select('name age profile \_id') // if we want to specific field

userModel.find().project('\_id') // if we want all field except \_id

project -> remove data from given result,

select -> which data we want to show.

@here

Hello everyone,

Today marks my last day, and I just wanted to take a moment to express my gratitude. It has been a truly wonderful experience working here, and I’ve enjoyed every moment (specially Friday). It has been a great journey working with such an amazing team, and I truly appreciate all the experiences that have helped me grow both professionally and personally.

I am especially grateful to my teammates and manager for their constant support and encouragement. Your guidance and camaraderie have made this journey memorable.

Wishing you all continued success, and I hope our paths cross again in the future!

Thank you once again for everything!

Best regards,

Pintoo Rana