**NODE JS**

**What is node JS?**

Node.js is an open-source, cross-platform JavaScript runtime environment that allows developers to run JavaScript code outside the browser, primarily on the server-side.

It is built on Google Chrome's V8 engine, which compiles JavaScript to native machine code, making it highly fast and efficient.

One of Node.js's biggest strengths is that it's **event-driven and non-blocking**, which allows it to handle thousands of concurrent connections using a single-threaded event loop. This makes it ideal for scalable, I/O-heavy, and real-time applications.

It's widely used to build the backend of web applications — such as RESTful APIs, chat apps, and data streaming services — which the frontend can communicate with via HTTP or WebSocket protocols.

Overall, Node.js provides the performance and flexibility needed for modern web development, especially when handling real-time data and high concurrency.

**How Node is a runtime environment on server side? What is V8?**

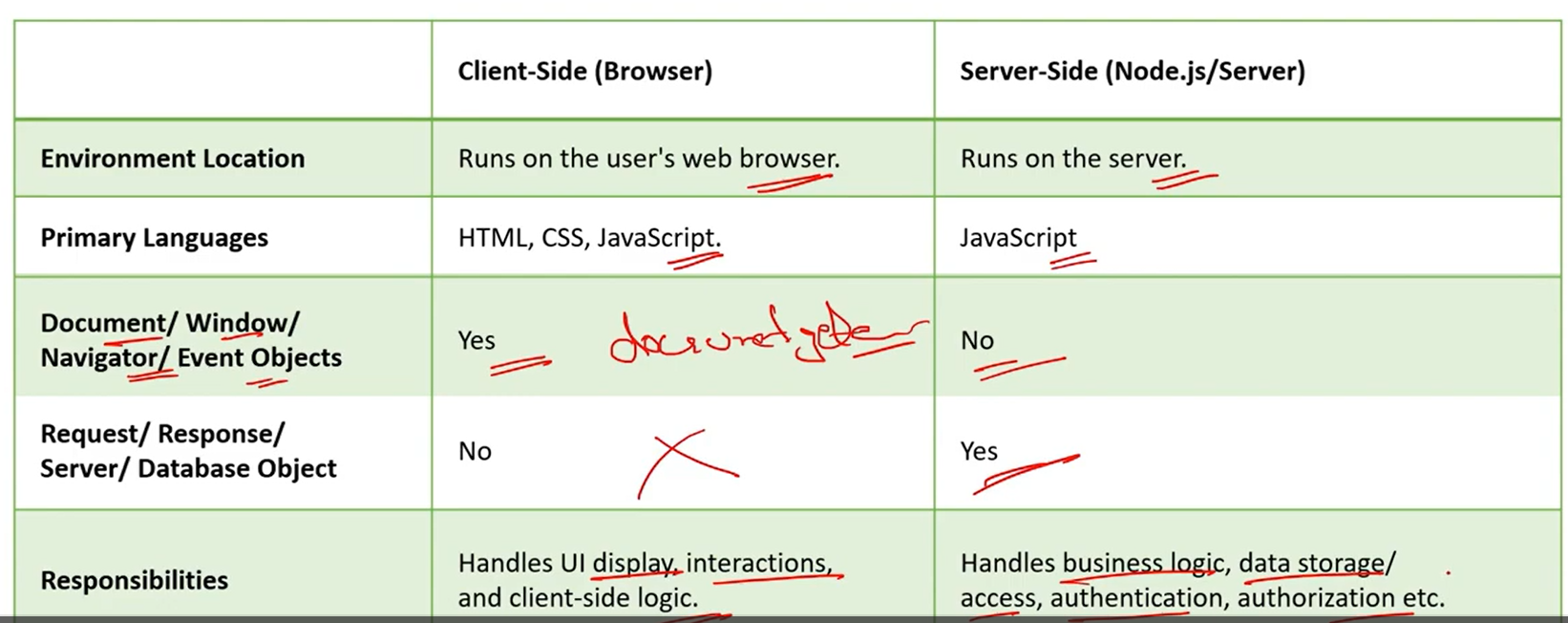
Javascript will execute in all browsers with help of search engine **(ex.V8)**

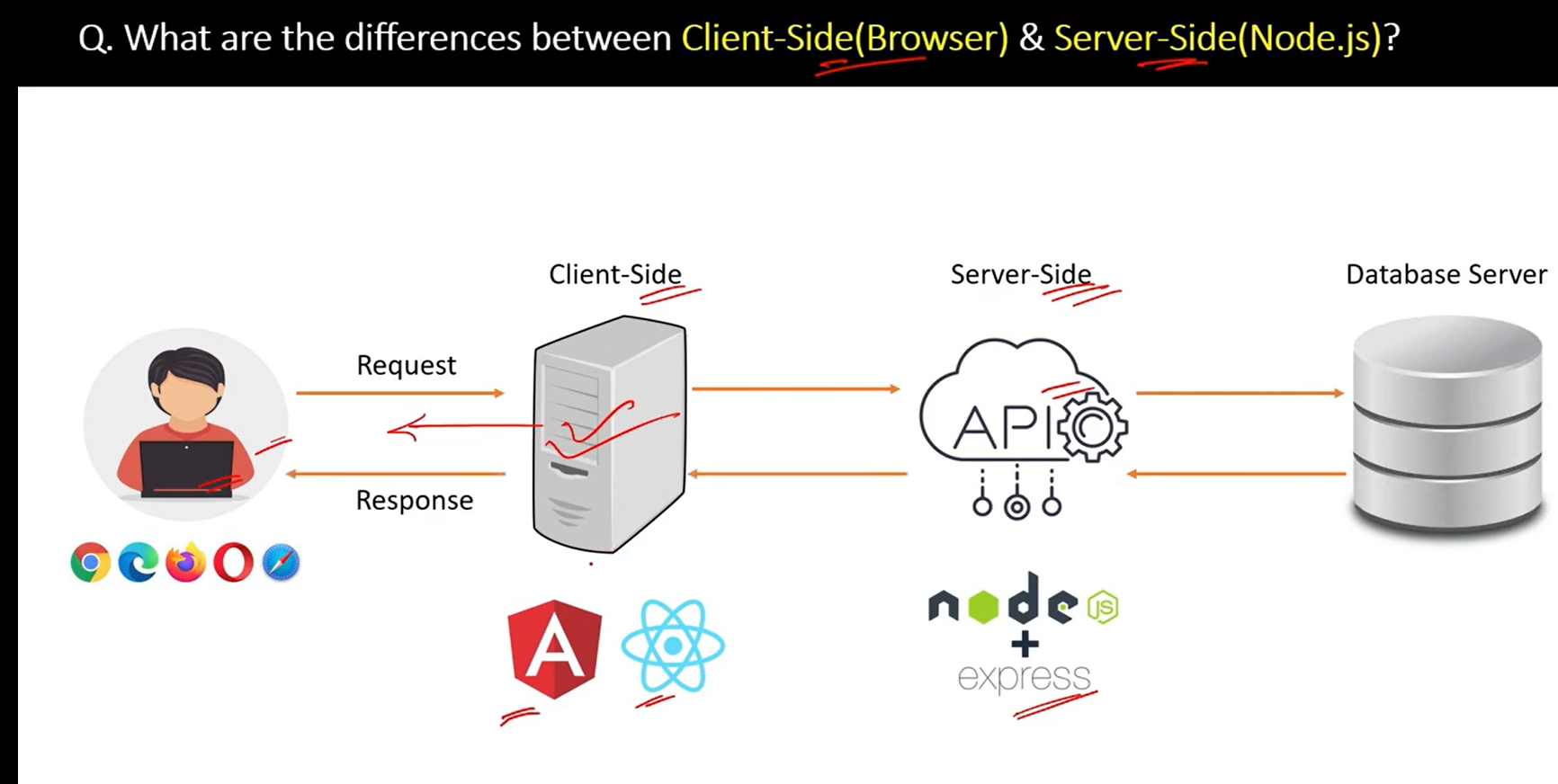
v8 is wrapped on node js so JavaScript will execute on server side.

**What is the difference between Runtime environment & Framework?**

**RUN TIME ENV:** primarily focus on providing necessary infrastructure for code execution.

**FRAME WORK:** simplifies of code of large applications by providing many inbuilt and simple routing system, middle ware support (**EXPRESS JS)**





**7 main features of NODE.jS?**

Single threaded

Asynchronous

Event-Driven

V8 Javascript engine

Cross platform

NPM(node package manager)

Real-time capabilities

It is javascript run time, if want to run JavaScript outside of browser NODE JS is required.

NODE JS have **v8 Engine** it will support javascript performance related things.

npm init-- (node setup will take place, package.json will create)

example of package.json

|  |
| --- |
| {  **"name"**: **"todo-backend"**,  **"version"**: **"1.0.0"**,  **"description"**: **"todo list project for learning"**,  **"main"**: **"index.js"**,-🡪**renamed as server.js, we can modify as per our wish**  **If used “type”:module**  **"scripts"**: {  **"test"**: **"echo** \"**Error: no test specified**\" **&& exit 1"**    },  **"author"**: **"jhana"**,    **"license"**: **"ISC"**,  **"dependencies"**: {-------🡪**npm install express, npm install mongoose**  **"express"**: **"^4.19.2"**,  **"mongoose"**: **"^8.5.3"**    }  } |

In js file.

* Have to import express const express = require('express');

We can import as import import *express* from **"express"**; **If used “type”:module**



* Instance of express need to store in variable. **const** *app* =  express();
* Define a route in order to access API usig url.

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

**res.send**(“hello World”)

})

const port = 3000;

app.listen(port, () => {

console.log(“server is listening”,+ port);

})

In terminal run : node server.js ------**(node js file name)**

**(npx kill port === to kill port)**

**//CREATING A TODO LIST**

**Let todos =[];**

***app*.post('/todos',(*req*,*res*)=>{**

**const {title, description} = *res*.*body*;---(where the output from API from todos url)**

**const *newTodo* ={// newTodo—storing of API values from todos**

**id : 0,**

***title*,**

***description***

**};**

***todos*.push(*newTodo*);pushing values to todos from newTodo**

***console*.log(*todos*);//logging it**

***res*.status(201).json(*newTodo*);//sending newTodo values JSON format**

**})**

**What is the difference between Node.js & Express.js?**

Node.js is a runtime environment that allows the execution of javascript outside of browser.(server side)

Express.js is a framework built on the top of the node.js

It is designed to simplify the process of building web applications and APIs

**What are the differences between Client-Side(Browser) & Server-Side(Node.js)?**

**SINGLE THREADING?**

When its get a request it creates single threaded to perform the operation and push it to the event loop to take the next operation.

It follows synchronous approach so each task is performed one after the other, program needs to wait for each operation to get complete before moving to other.

Synchronous programming focuses on the order of execution in the sequential manner with event loop module to handle multiple concurrency.

It used Portable Operating system interface for various I/O operations.

**Synchronous programming?**

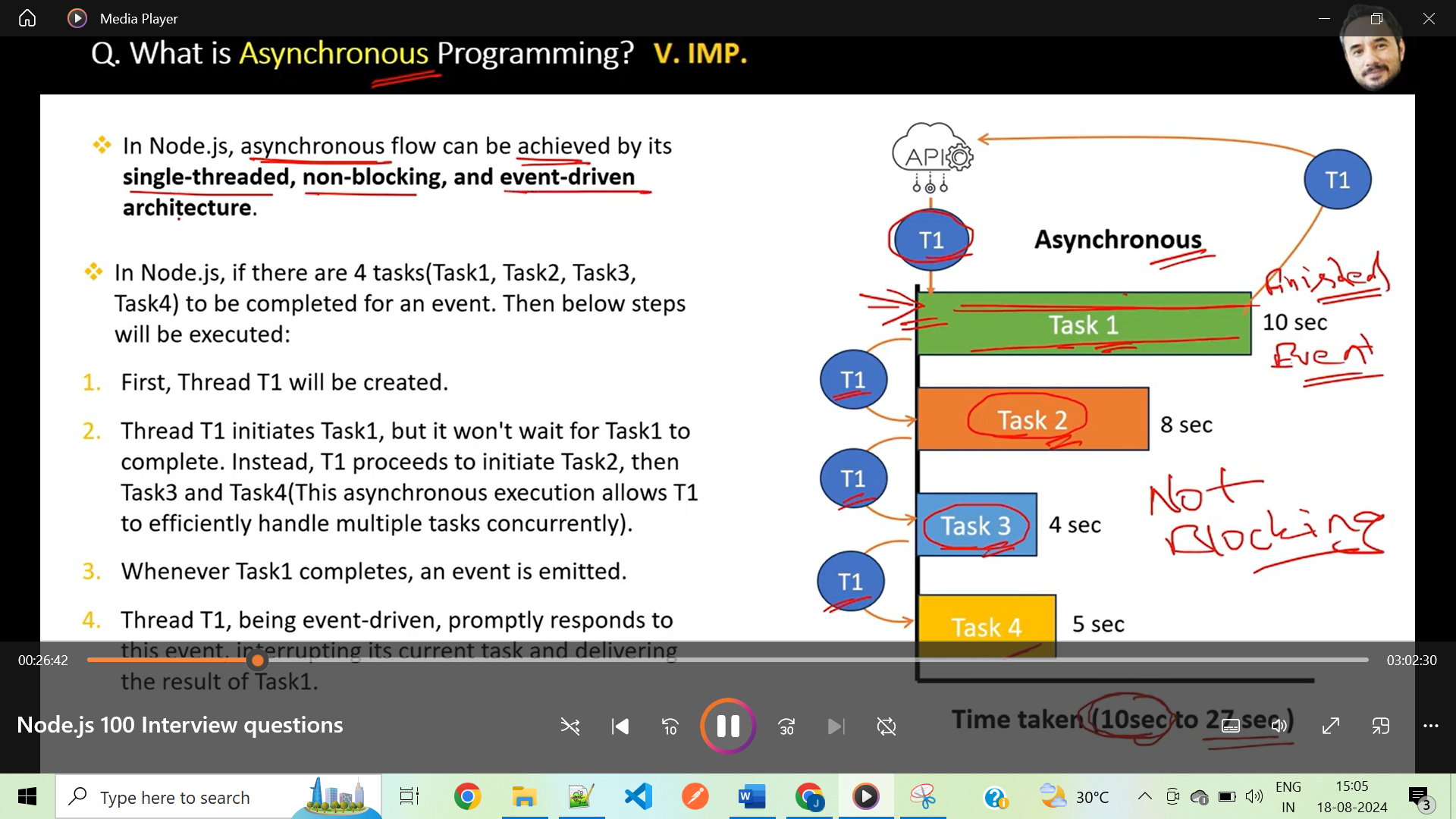
Until the previous execution get finished machine needs to hold onto memory and processing time for thread is idle.

**Multiple Threading?**

Once first task created second task will start with another thread.

**Asynchronous?**

**Asynchronous** means that code runs **without blocking** the execution of other operations. In Node.js, this allows the application to handle **multiple tasks simultaneously**, even though it runs on a **single thread**.

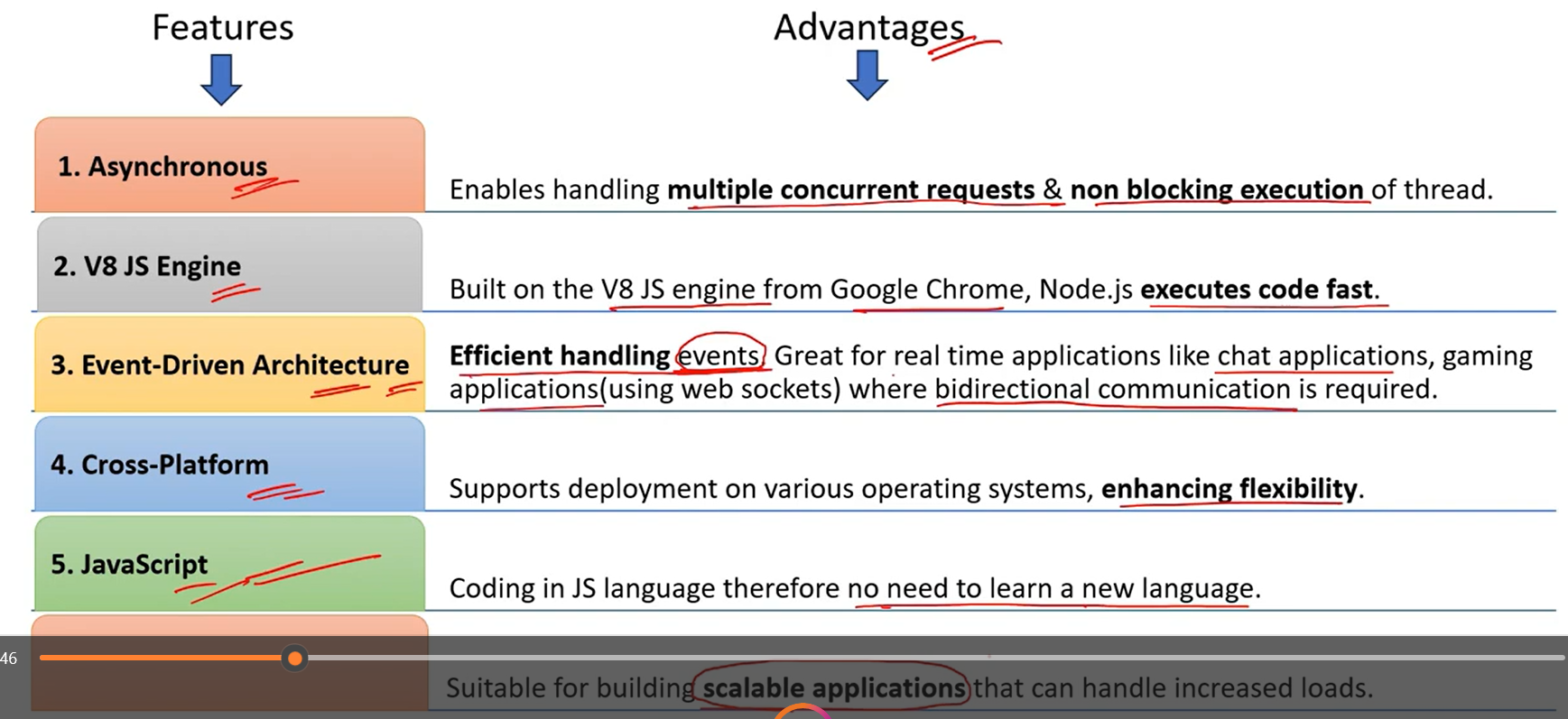


**Difference between Synchronous and asynchronous?**



**What are the main features of Node.js?**

**Advantages and features of Node JS?**



**What are the disadvantages of node?**

Node.js uses a single-threaded event loop, which is great for I/O tasks, but not suitable for CPU-intensive operations like image processing or complex calculations. Such tasks can block the event loop and degrade performance for all users.

Though Promises and async/await have improved this, older Node.js code often suffers from **deeply nested callbacks**, making the code harder to maintain and debug.

The npm ecosystem is huge, but some packages lack **proper documentation, testing, or maintenance**, which can lead to security or reliability issues if you're not careful.

In asynchronous environments, **missing error handling in one part of the code** (like a rejected Promise or stream error) can crash the entire application if not managed properly.

**Explain the difference between CommonJS and ES Modules in Node.js?**

Node.js supports two module systems for managing imports and exports:

1. CommonJS (CJS) – the original module system in Node.js
2. ES Modules (ESM) – the modern, standard module system from JavaScript (ECMAScript)

**What is clustering in Node.js and when should you use it?**

**Clustering** in Node.js is a technique that allows you to **create multiple worker processes** (child processes) to handle the load of a Node.js application, thereby enabling **multi-core CPU utilization**.

By default, Node.js runs on a **single thread**, which means it uses only **one CPU core**, regardless of how many are available. Clustering helps scale the app across **multiple cores**.

**How do you design scalable RESTful APIs?**

Designing **scalable RESTful APIs** means creating APIs that can handle **increasing load**, **users**, and **data** without breaking or slowing down. Scalability isn't just about adding servers—it's about **smart design choices** from the beginning.

**What are the key principles of REST architecture?**

**REST (Representational State Transfer)** is an architectural style for designing **networked applications**, especially web APIs. It was introduced by **Roy Fielding** in his PhD dissertation.

**How do you handle API versioning?**

API versioning is the process of managing changes to your API without breaking existing clients. As your application evolves, versioning ensures backward compatibility and smooth transitions for consumers.

**How would you implement rate limiting in an API?**

**Rate limiting** is a technique used to **control the number of requests** a client can make to your API in a given timeframe. It protects your server from abuse, DoS attacks, and helps ensure fair usage among clients.

**Explain the importance of idempotency in HTTP methods?**

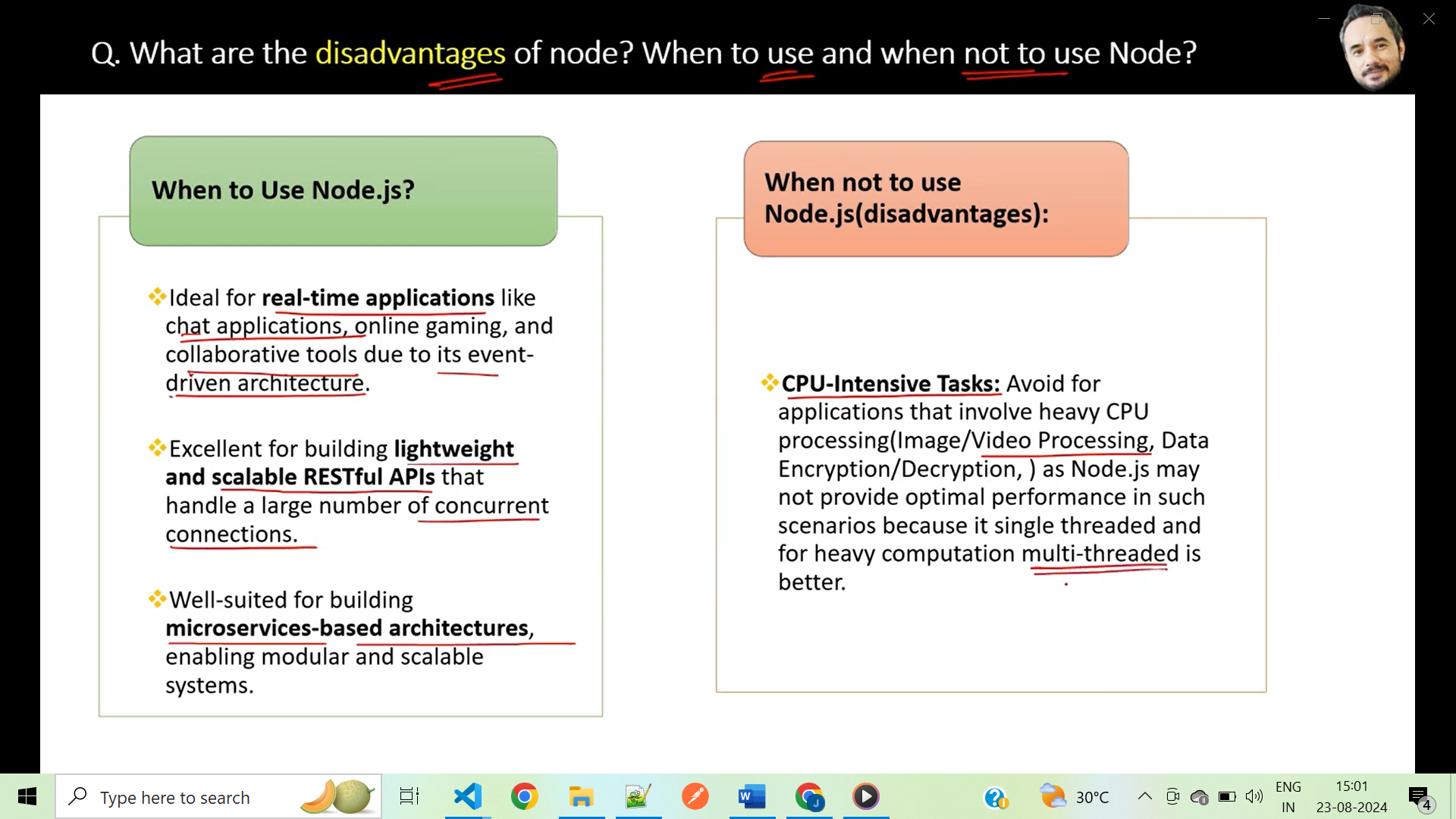
**Idempotency** means that **making the same HTTP request multiple times** has the **same effect** as making it **once**. In other words:

**An idempotent operation does not change the result beyond the initial application.**

**What is the role of the libuv library in Node.js?**

libuv is a C-based multi-platform support library that provides Node.js with asynchronous I/O capabilities. It’s the backbone of Node.js's non-blocking architecture and powers the event loop, thread pool, and asynchronous operations.

**When to use and when not to use Node?**



**How to setup node.js project?**

**What is NPM? What is the role of node\_modules folder?**

**What is the role of package.json file in Node?**

**STREAMS?**

**Streams** in Node.js are **objects** that let you **read or write data piece by piece (chunks)** instead of loading the entire data into memory.

This makes them highly efficient for handling **large files**, **network operations**, or **real-time data processing** — as they reduce **memory consumption** and improve **performance**.

Node.js provides four main types of streams:

* Readable – for reading data (e.g., fs.createReadStream)
* Writable – for writing data (e.g., fs.createWriteStream)
* Duplex – both readable and writable (e.g., TCP sockets)
* Transform – modifies data while reading/writing (e.g., compression)

Imagine watching a video on YouTube — it doesn’t download the entire video first, it **streams** it bit by bit. Similarly, Node.js uses **streams** to process big files or data without blocking the application



**What is buffers in Node.JS?**

Buffer is a temporary memory used in streams to hold on the data until consumed.

Used to represent a fixed length sequence of bytes.

It is fixed allocation memory outside v8 engine.

**What is REPL in Node.JS?**

**Read Even Print Loop** it is a program that accepts commands.

Read user inputs, parses the data into JS data-structure and stores in a memory.

**What is NPM? What is the role of node\_modules folder?**

**What is the role of package.json file in Node?**

**What is the Event Loop in Node.js? Explain with an example**

Node.js is **single-threaded** and uses the **Event Loop** to handle **asynchronous operations** (like file reads, API calls, DB queries) **without blocking** the main thread.

Think of the event loop as a **manager** that continuously checks if there is anything to do (like callbacks, timers, I/O). It runs in **phases**, and each phase handles specific types of tasks.

Event loops is an endless loop, which waits for incoming tasks, executes them and then sleeps until receives more tasks, it allows to use javascript outside of the browser.

Event Loop is **core to Node.js's non-blocking behavior**.

It handles async code like setTimeout, Promises, and I/O.

process.nextTick() is a **microtask**, runs before Event Loop phases.

setImmediate() runs in **Check phase**.

It makes Node.js scalable by not blocking the main thread.

**Event, Event Emitters, Event Queue, Event Loop Event Driven?**

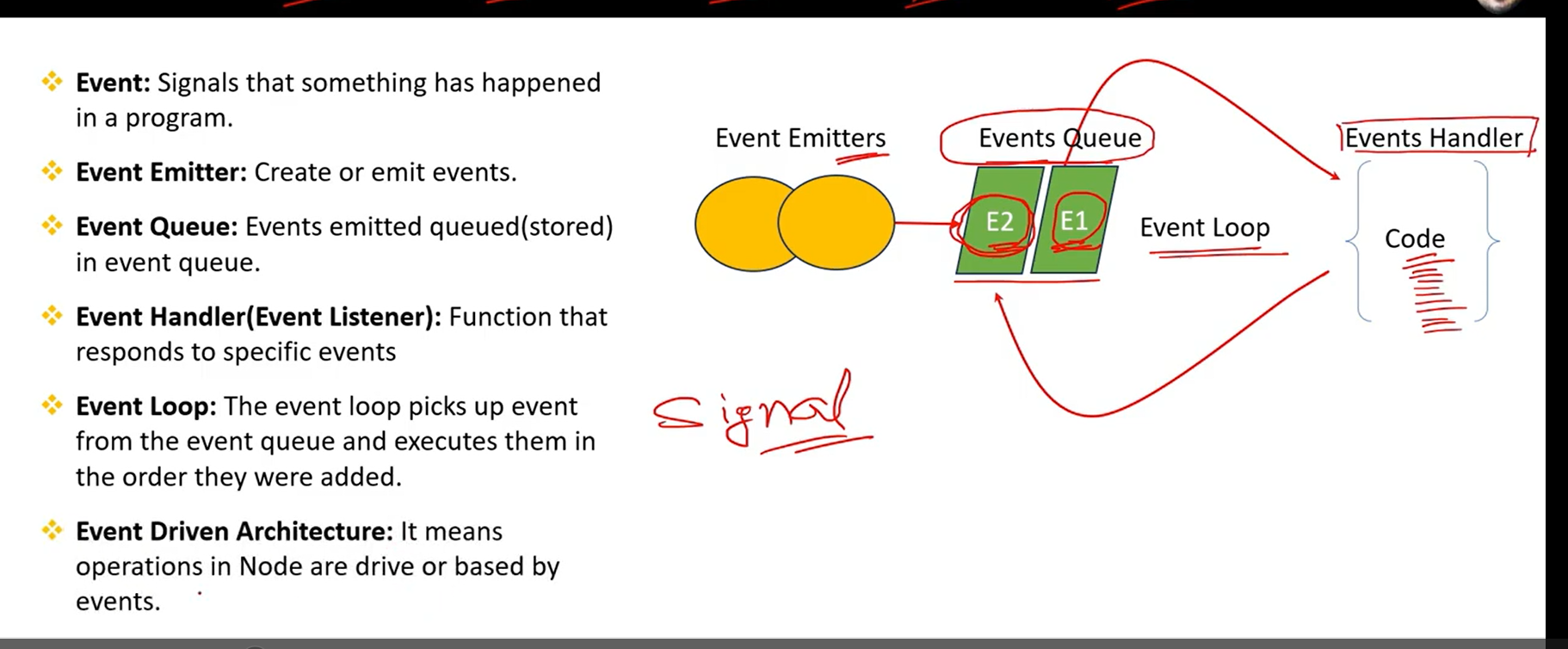
Event: Any action occurred the signal will trigger/signal that something happened in a program .

Event Emitters: The source of action is called event emitters.(create or emit the events)

Event Queue: All generated events are stored in a queue.

Event Loop: Picking up of events from event queue in same order is called

Event Driven

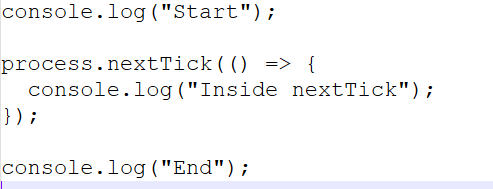


**What is the difference between process.nextTick(), setImmediate(), and setTimeout() in Node.js?**

In Node.js, all three — process.nextTick(), setImmediate(), and setTimeout() — are used to schedule asynchronous operations, but they work at different phases of the event loop.

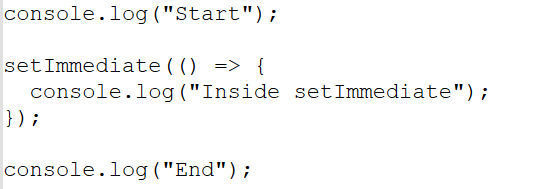
**process.nextTick(callback)**

* Executes **after the current operation** completes and **before** the next event loop tick starts.
* It has **higher priority** than other timers.
* Used when you want to **delay execution of a function** until the call stack is cleared but **before** I/O events.



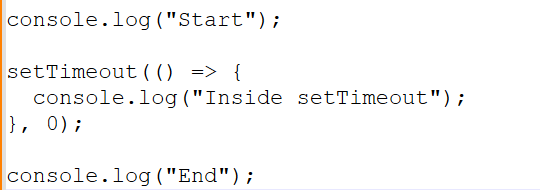
**setImmediate(callback)**

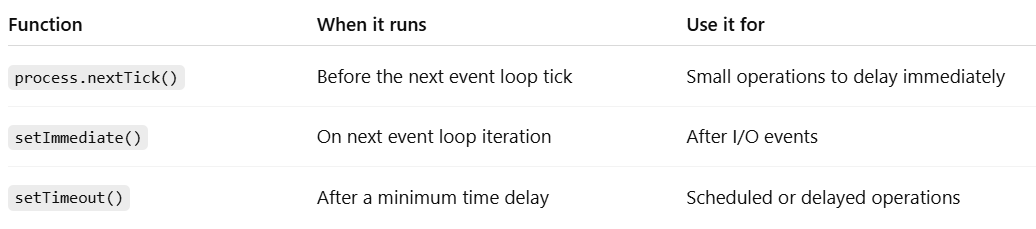
* Executes the callback **on the next iteration** (or tick) of the event loop.
* It executes **after I/O events**.



**setTimeout(callback, delay)**

* Executes the callback **after at least the given delay** (in ms).
* The actual delay depends on system load.





**What is Error Handling? In how many ways you can do error handling in Node.js?**

Error handling in Node.js is the process of catching and managing unexpected issues that occur during runtime — like failed API calls, invalid user inputs, or file system errors.

Node.js follows a non-blocking asynchronous model, so errors can occur both in synchronous and asynchronous code. To handle them, I typically use:

1. Try...Catch – For synchronous errors.
2. Callbacks with Error-first Convention – Common in core modules. The first parameter is usually err, like in fs.readFile().
3. Promises with .catch() – When using async code with promises.
4. Async/Await with Try...Catch – A cleaner way to handle asynchronous errors.

**What is Error-First Callbacks?**

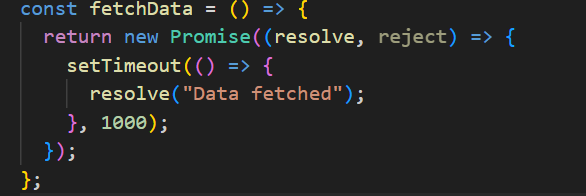
**What is a callback and callback hell in Node.js?**

A **callback** is a function passed as an argument to another function and is executed after the parent function completes its operation — especially useful in asynchronous programming like reading files or calling APIs in Node.js.

**Callback hell** occurs when many asynchronous operations are nested inside callbacks, leading to deep indentation, complex logic, and difficult-to-maintain code.

**Explain callback vs Promise vs async/await in Node.js?**

is an object representing the eventual completion (or failure) of an asynchronous operation. It helps manage async code more cleanly compared to nested callbacks, avoiding **callback hell**. To improve code structure and error handling, Promises were introduced, allowing better chaining through .then() and .catch().



1. **Pending** – Initial state, neither fulfilled nor rejected.
2. **Fulfilled** – Operation completed successfully.
3. **Rejected** – Operation failed.

As the ecosystem matured, async/await—introduced in ES2017—became the preferred approach. It enables writing asynchronous code that reads like synchronous code, significantly improving **readability**, **debuggability**, and **team maintainability**. From a leadership standpoint, adopting async/await ensures cleaner control flow, better onboarding for new developers, and fewer logic bugs, especially in full-stack systems involving multiple I/O operations like DB, file handling, and external APIs.

**How is a Promise different from a callback?**

* Callbacks can lead to **nested chains** (callback hell), making code harder to read and maintain.
* Promises offer **chaining using .then()** and better **error handling with .catch()**.
* Promises make async code more manageable and readable.

**How to handle errors using Promises?**

Use .catch() to handle rejections.

|  |
| --- |
| fetchData()  .then((data) => console.log(data))  .catch((error) => console.error("Error:", error)); |

**What is Promise chaining?**

Chaining allows multiple asynchronous operations to run sequentially using .then().

|  |
| --- |
| doStep1()  .then(doStep2)  .then(doStep3)  .catch(handleError); |

Each .then() returns a new Promise.

**What is Promise.all() and when do you use it?**

Promise.all() runs multiple promises **in parallel** and waits for all to resolve. It fails fast if **any** promise rejects.

|  |
| --- |
| Promise.all([fetchUser(), fetchPosts()])  .then(([user, posts]) => {  console.log(user, posts);  })  .catch(err => console.error("Error:", err)); |

**Difference between Promise.all(), Promise.race(), and Promise.any()**

**Promise.all():** Waits for all to resolve, rejects if any fails.

**Promise.race():** Returns result of the first settled promise (resolve or reject).

**Promise.any():** Returns the first fulfilled promise, ignores rejections.

**How to handle errors while using async-await?**

|  |
| --- |
| async function loadData() {  try {  const result = await fetchData();  console.log(result);  } catch (err) {  console.error("Error:", err);  }  } |

**How to handle errors in synchronous operations using try-catch-finally?**

**What are Modules in Node? What is the difference between a function & module?**

Module is a specific functionality that can be reused with in node.js application.

Javascript file will be treated as module in Node .JS

Module contain multiple function, objects and variables.

By default it is not accessible outside of file, inorder to access need to export it.

Module.export.functionName= functionName

**Inbuilt function: const module1 = requires(‘./moduele’);**

Each module wrapper in a function called module wrapper function.

**What is fsmodule? And its function.**

It is used to manage files and provides set of methods for interacting with the file system.

**How many ways are there to Export a module?**

**What will happen if you don’t export the module?**

**How to import single and multiple functions from a module?**

**What is module wrapper function?**

**What are the Types of modules in Node?**

**What are the Top 5 built in modules commonly used in node projects?**

**Explain the role of fs module? Name some functions of it?**

**Explain the role of path module? Name some functions of it?**

**Explain the role of OS module? Name some functions of it?**

**Explain the role of events module? How to handle events in Node?**

**What are Event Arguments?**

**What is the difference between a function and an event?**

**What is the role of http module in node?**

**What is the role createServer() method of http module?**

**What are the advantages of using Express.js with Node.js?**

**How do you install Express.js in a Node.js project?**

**How to create an HTTP Sever using Express.js?**

**How do you create and start Express.js application?er**

**What is Middleware in Express.js and when to use them?**

middleware is a function that runs before the final route handler. It has access to the request (req), response (res), and the next() function. Middleware is used to execute code, modify req or res, end the request-response cycle, or pass control to the next middleware.

**When to Use Middleware:**

* **Before protected routes** (e.g., checking JWT or roles)
* **Global error handling**
* **Preprocessing requests** (parsing body, sanitizing inputs)
* **Serving static files**

**How do you implement middleware in Express.js?**

To implement middleware in Express.js, I create a function with three parameters: req, res, and next. Inside the function, I can run logic (e.g., auth check, logging), modify the request or response objects, and then call next() to pass control to the next middleware or route. I register middleware using app.use() or directly inside a route.

**What is the purpose of the app.use() function in Express.js?**

app.use() is used to register middleware in Express.js. It applies the middleware globally to all incoming requests or to a specific path prefix. It’s commonly used for logging, authentication, request parsing, and error handling.

**What is the purpose of the next parameter in Express.js?**

In Express.js, the next parameter is a function that passes control to the next middleware or route handler in the stack. Without calling next(), the request will hang and not proceed. It's essential for building layered logic like authentication, logging, and error handling.

If you forget to call next() in async middleware or after a condition, the request will hang indefinitely. Always ensure it's called properly

**How to use middleware globally for a specific route?**

**What is Request Pipeline in Express?**

**What are the Types of middleware's in Express.js?**

There are 4 main types of middleware in Express:

1. Application-level middleware – tied to an app (app.use() or app.get())
2. Router-level middleware – used with express.Router()
3. Built-in middleware – like express.static, express.json
4. Error-handling middleware – has 4 parameters: (err, req, res, next)

**What is the difference btw application-level & route-level middleware?**

**What is error handling middleware and how to implement it?**

**If you have 5 middlewares then in which middleware you will do the error handling?**

**What is built in middleware? How to serve static files from Express.js?**

**What is third-party middleware's? Give some examples?**

**Can you summarize all the type of middleware's?**

**What are the advantages of using middleware in Express.js?**

**What is Routing in Express.js?**

**What is the difference between middleware & routing in Express?**

**How to implement routing? How do you define routes in Express.js?**

**How to handle Routing in Express.js real applications?**

**What are Route Handlers?**

**What are Route Parameters in Express.js?**

**What are Router object & Router Methods and how to implement them?**

**What are the types of Router Methods?**

**What is the difference between app.get() and router.get() method?**

**What is express.Router() in Express.js?**

**Share a real application use of Routing?**

**What is Route Chaining in Express.js?**

**What is Route Nesting in Express.js?**

**How to implement route nesting in Express.js?**

**What are Template Engines in Express.js?**

**Name some Template Engines libraries?**

**How to implement EJS templating engine in a Express.js application?**

**What is REST & RESTful API?**

**What are HTTP Request and Response structures in UI and REST API?**

**What are Top 5 REST guidelines and the advantages of them?**

**What is the difference between REST API and SOAP API?**

**What are HTTP Verbs and HTTP methods?**

**What are GET, POST, PUT & DELETE HTTP methods?**

**What is the difference between PUT & PATCH methods?**

**Explain the concept of Idempotence in RESTful APIs.**

**What are the role of status codes in RESTful APIs?**

**What is CORS in RESTful APIs?**

**How to remove CORS restrictions RESTful APIs?**

**What are Serialization & Deserialization?**

**What are the types of serialization?**

**How to Serialize & Deserialize in Node.js?**

**Explain the concept of versioning in RESTful APIs.**

**What is an API document? What are the popular documentation formats?**

**What is the typical structure of a REST API project in Node?**

**What are Authentication and Authorization?**

These are two critical concepts in security and access control for APIs and web applications:

1. Authentication (Who are you?)

Definition: The process of verifying a user’s identity.

Example: Logging in with a username and password.

Result: If successful, the system knows who the user is.

2. Authorization (What are you allowed to do?)

Definition: The process of checking whether the authenticated user has permission to access a resource or perform an action.

Example: A regular user may be denied access to the admin dashboard.

**Simple Analogy:**

Authentication is showing your ID card at the gate.  
Authorization is whether you're allowed into the VIP room inside.

**What are the types of authentication in Node.js?**

Node.js supports various authentication methods, commonly through packages like passport, jsonwebtoken, or custom middleware.

Basic Authentication

Token-Based Authentication (e.g., JWT)

OAuth 2.0

Session-Based Authentication

API Key Authentication

Third-party logins (e.g., Google, GitHub)

**What is Basic Authentication?**

Basic Auth is a simple authentication method built into the HTTP protocol.

**The client sends a request with the Authorization header:**

Authorization: Basic base64(username:password)

**The server decodes it and validates the credentials.**

GET /dashboard

Authorization: Basic dXNlcjEyMzpwYXNzMTIz

**How do you design scalable RESTful APIs?**

To design scalable RESTful APIs, you should focus on principles that improve performance, maintainability, security, and extensibility:

🔹 Key Practices:

* Use proper HTTP methods: GET, POST, PUT, DELETE, etc.
* Resource-oriented URIs: /users/123, not /getUser?id=123
* Statelessness: Each request must contain all information needed (no session dependency).
* Pagination & Filtering: For large datasets, use ?page=1&limit=20
* Rate limiting & caching: Use ETag, Cache-Control, Redis, etc.
* Asynchronous processing: Use queues (e.g., RabbitMQ) for heavy tasks.
* Horizontal scalability: Stateless design helps scaling with load balancers.
* Validation & error handling: Standard response formats (like JSON:API) with clear HTTP status codes.

**GET /api/v1/products?page=2&limit=10&sort=price**

## What are the key principles of REST architecture?

## REST stands for Representational State Transfer and follows 6 architectural principles:

| Principle | Description |
| --- | --- |
| 1. Client-Server | UI (frontend) and API (backend) should be separate. |
| 2. Stateless | Server does not store any session data; each request is independent. |
| 3. Cacheable | Responses must define cacheability (Cache-Control, ETag). |
| 4. Uniform Interface | Consistent URLs, HTTP methods, media types (JSON, XML). |
| 5. Layered System | Multiple layers (e.g., API Gateway, microservices) should behave uniformly. |
| 6. Code on Demand (optional) | Clients can download code like JS for execution. |

## How do you handle API versioning?

API versioning allows you to introduce changes without breaking existing clients.

🔹 Common Strategies:

| Method | Example | Notes |
| --- | --- | --- |
| URI Path | /api/v1/users | Most common and visible |
| Request Header | Accept: application/vnd.api.v1+json | Clean URL, but hidden |
| Query Param | /users?version=1 | Less common, not REST-pure |

Best Practice: Use URI versioning (/v1) for public APIs.

app.use('/api/v1', require('./routes/v1/users'));

app.use('/api/v2', require('./routes/v2/users'));

## Explain the Node.js event loop and how it works.

The Node.js event loop is the core mechanism that handles non-blocking I/O operations despite Node.js running on a single-threaded event-driven architecture. It enables asynchronous programming by offloading operations (e.g., file I/O, network) to the libuv thread pool and executing callbacks once those operations complete.

How it works (phases):

Timers Phase – Executes callbacks from setTimeout() and setInterval().

Pending Callbacks Phase

Idle/Prepare Phase

Poll Phase – Waits for I/O events.

Check Phase – Executes setImmediate() callbacks.

Close Callbacks Phase

|  |
| --- |
| setTimeout(() => console.log("Timeout"), 0);  setImmediate(() => console.log("Immediate"));  process.nextTick(() => console.log("NextTick"));  // Output:  // NextTick  // Immediate or Timeout (depends on context) |

## What is the difference between process.nextTick(), setImmediate(), and setTimeout()?

- process.nextTick(): Executes after the current operation completes but before any I/O events.  
- setImmediate(): Executes after the current event loop phase completes.  
- setTimeout(): Executes after a minimum threshold (ms), not guaranteed immediately.

|  |
| --- |
| setTimeout(() => console.log('setTimeout'), 0);  setImmediate(() => console.log('setImmediate'));  process.nextTick(() => console.log('nextTick'));  // Output: nextTick → setTimeout or setImmediate (depends) |

## How does Node.js handle asynchronous operations?

Node.js uses **event-driven**, **non-blocking I/O** powered by the **event loop** and **libuv**. When an async operation (e.g., fs.readFile) is initiated, it's offloaded to the libuv thread pool or system kernel. Once the task completes, the associated callback is queued in the event loop for execution.

**Mechanisms:**

Callbacks, Promises, Async/Await, Event Emitters, Streams

|  |
| --- |
| const fs = require('fs/promises');  async function readFile() {  try {  const data = await fs.readFile('test.txt', 'utf8');  console.log(data);  } catch (err) {  console.error(err);  }  }  readFile(); |

## What is the role of the libuv library in Node.js?

libuv is a **C++ library** that powers Node.js's asynchronous behavior. It provides:

A **thread pool** for file system & DNS operations.

**Event loop implementation**

Cross-platform **non-blocking I/O** APIs

Handles **TCP/UDP**, **timers**, **pipes**, **signals**, etc.

**Key Roles:**

Bridges Node.js and OS

Makes single-threaded async I/O possible

**Example:**  
When you call fs.readFile(), libuv delegates the task to a thread pool, allowing Node.js to continue processing other requests.

**Explain the difference between CommonJS and ES Modules in Node.js.**

- CommonJS: `require()` and `module.exports`; used by default in Node.js.  
- ES Modules: `import` and `export`; standardized in ECMAScript.  
  
Example:  
```js  
// CommonJS  
const fs = require('fs');  
module.exports = {};  
  
// ES Module  
import fs from 'fs';  
export default {};  
```

## What is clustering in Node.js and when should you use it?

Clustering allows you to create child processes (workers) to handle load across multiple CPU cores. Useful for improving performance of compute-heavy or I/O-bound operations.  
  
Example:  
```js  
const cluster = require('cluster');  
if (cluster.isMaster) {  
 cluster.fork();  
} else {  
 // Worker code  
}  
```

**What are the security risks associated with storing passwords in plain text in Node.js?**

**What is the role of Hashing and Salt in securing passwords?**

**How can we create hash passwords in Node.js?**

**What is API Key Authentication?**

**What is Token based and JWT authentication?**

**What are the parts of JWT token?**

**Where JWT token reside in the request?**

**How can you Debug Node.js applications?**

**What is XSS (Cross-Site Scripting) attack?**

**How to prevent XSS attacks? What is sanitize-html module?**

**What is SQL Injection attack?**

**How do you prevent SQL injection attacks in Node?**

**How can you improve the performance of a Node.js application?**

**How can you Deploy a Node.js application?**

**What is Testing? What are the types of testing?**

**What are testing frameworks?**

**How to create unit tests using Jest framework?**

**How to write code for Unit Testing? Explain AAA?**

**What is Test-Driven Development (TDD)?**

**What is WebSocket? How to create a real-time chat application using Node?**

**Core Node.js Concepts**

**- Explain the Node.js event loop and how it works.**

**- What is the difference between process.nextTick(), setImmediate(), and setTimeout()?**

**- How does Node.js handle asynchronous operations?**

**- What is the role of the libuv library in Node.js?**

**- Explain the difference between CommonJS and ES Modules in Node.js.**

**API Design & RESTful Services**

**- How do you design scalable RESTful APIs?**

**- What are the key principles of REST architecture?**

**- How do you handle API versioning?**

**- How would you implement rate limiting in an API?**

**- Explain the importance of idempotency in HTTP methods.**

**Authentication & Security**

**- How do you implement authentication and authorization in Node.js?**

**- What is the difference between JWT and session-based authentication?**

**- How do you protect Node.js apps from common security threats (e.g., XSS, CSRF, SQL injection)?**

**- What is helmet.js and how does it improve security?**

**- How do you securely store and manage API keys and secrets?**

**- Compare SQL and NoSQL databases. When would you use each?**

**- How do you handle transactions in MongoDB or PostgreSQL with Node.js?**

**- What are ORMs and when should you use them (e.g., Sequelize, TypeORM)?**

**- How do you implement caching in a Node.js application?**

**- How do you handle database migrations and schema versioning?**

**Performance & Optimization**

**- What tools do you use for profiling and monitoring Node.js applications?**

**- How do you handle memory leaks in Node.js?**

**- What is backpressure in streams and how do you manage it?**

**- Explain the use of load balancers with Node.js applications.**

**- How would you optimize response times for high-traffic APIs?**

**Testing & CI/CD**

**- What are the best practices for unit and integration testing in Node.js?**

**- Which testing frameworks do you use (e.g., Mocha, Jest, Supertest)?**

**- How do you mock external APIs in tests?**

**- How do you set up a CI/CD pipeline for a Node.js backend?**

**- What strategies do you use for database testing?**

**Architecture & System Design**

**- How would you design a scalable microservices architecture with Node.js?**

**- What are the benefits and drawbacks of monolith vs microservices?**

**- How do you manage service-to-service communication (e.g., gRPC, REST, message queues)?**

**- What is your approach to designing a logging and monitoring strategy?**

**- How do you ensure fault tolerance and high availability?**

**DevOps & Deployment**

**- How do you containerize a Node.js app using Docker?**

**- What is your approach to environment configuration and secrets management?**

**- How do you use PM2 for process management?**

**- How do you deploy Node.js applications on cloud platforms (AWS, GCP, Azure)?**

**- What is blue-green deployment and how do you implement it?**

**How does Node.js differ from traditional web servers like Apache?**

Node.js uses an event-driven, non-blocking I/O model which makes it lightweight and efficient. Unlike Apache, which creates a new thread for each request, Node.js handles multiple requests on a single thread using asynchronous callbacks.

**Explain the characteristics of Node.js that make it suitable for building scalable applications.**

Non-blocking I/O: Handles many connections concurrently.

Event-driven architecture: Efficiently handles asynchronous operations.

Single-threaded: Uses fewer resources and is easier to manage.

**What is npm? How do you use it in Node.js projects?**

npm (Node Package Manager) is a package manager for JavaScript, included with Node.js. It helps install, update, and manage packages.

Example:

// Initialize a new Node.js project

npm init -y

// Install a package (e.g., Express)

npm install express

**What are the advantages of using Node.js?**

High performance due to V8 engine.

Efficient handling of concurrent connections.

Large ecosystem with npm.

Full-stack JavaScript development.

**What is event-driven programming in Node.js?**

Event-driven programming in Node.js means that the flow of the program is determined by events such as user actions, sensor outputs,or messages from other programs.

Example:

const EventEmitter = require('events');

const emitter = new EventEmitter();

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

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

});

emitter.emit('event');

7. **How do you include external libraries in Node.js?**

External libraries are included using the

require function.

Example:

const express = require('express');

const app = express();

8. **What is the difference between require and import in Node.js?**

require is used in CommonJS module system, while import is used in ES6

modules. Node.js natively supports CommonJS, but ES6 modules can be used with the .mjs extension or with specific

configurations.

Example:

// CommonJS

const express = require('express');

// ES6

import express from 'express';

9. **How do you create and publish your own npm package?**

Create a package.json file, write your code, and use npm publish to publish the package.

Example:

// Initialize project

npm init

// Create main file (index.js)

echo "module.exports = () => console.log('Hello,

world!');" > index.js

// Publish the package

npm publish

10. **What are the built-in modules in Node.js?**

Some built-in modules include http, fs, path, url, crypto, and events.

Example:

const fs = require('fs');

fs.readFile('file.txt', 'utf8', (err, data) => {

if (err) throw err;

console.log(data);

});

11. **Explain the concept of non-blocking I/O in Node.js.**

Non-blocking I/O means that operations like reading from a file or database do not block the execution of other operations. Instead, they are executed asynchronously, allowing other code to run concurrently.

12. **How does Node.js handle asynchronous operations?**

Node.js uses callbacks, Promises, and async/await to handle asynchronous

operations.

13. **What are callbacks in Node.js? How do you handle errors with callbacks?**

Callbacks are functions passed as arguments to other functions to be executed after the completion of an operation.

Example:

const fs = require('fs');

fs.readFile('file.txt', 'utf8', (err, data) => {

if (err) {

console.error('Error reading file:', err);

return;

}

console.log('File data:', data);

});

○

14. **Explain Promises in Node.js.**

Promises represent the eventual completion (or failure) of an asynchronous operation and its resulting value.

Example:

const fs = require('fs').promises;

fs.readFile('file.txt', 'utf8')

.then(data => console.log('File data:', data))

.catch(err => console.error('Error reading

file:', err));

15. **What are async/await in Node.js? How do they work?**

async/await is syntactic sugar built on top of Promises, making asynchronous code look and behave more like synchronous code.

Example:

const fs = require('fs').promises;

async function readFile() {

try {

const data = await fs.readFile('file.txt',

'utf8');

console.log('File data:', data);

} catch (err) {

console.error('Error reading file:', err);

}

}

readFile();

16. **How do you perform file operations in Node.js?**

Using the fs module for synchronous and asynchronous file operations.

Example:

const fs = require('fs');

// Asynchronous read

fs.readFile('file.txt', 'utf8', (err, data) => {

if (err) throw err;

console.log(data);

});

// Synchronous read

const data = fs.readFileSync('file.txt',

'utf8');

console.log(data);

17. **Explain the difference between synchronous and asynchronous file operations in Node.js.**

Synchronous operations block the event loop until the operation completes, while asynchronous operations do not block the event loop and use callbacks, Promises, or async/await for the result.

18. **How do you handle streams in Node.js?**

Streams are used to handle reading and writing of large data efficiently. They are instances of EventEmitter.

Example:

const fs = require('fs');

const readStream =

fs.createReadStream('largefile.txt', 'utf8');

const writeStream =

fs.createWriteStream('output.txt');

readStream.on('data', chunk => {

writeStream.write(chunk);

});

readStream.on('end', () => {

writeStream.end();

});

19. **How do you create a simple HTTP server in Node.js?**

Using the http module.

Example:

const http = require('http');

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

res.statusCode = 200;

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

res.end('Hello, World!\n');

});

server.listen(3000, '127.0.0.1', () => {

console.log('Server running at

http://127.0.0.1:3000/');

});

**20. What is Express.js? How do you use it in Node.js applications?**

Express.js is a minimal and flexible Node.js web application framework that provides a robust set of features for building web and mobile applications.

Example:

const express = require('express');

const app = express();

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

res.send('Hello, World!');

});

app.listen(3000, () => {

console.log('Server running on port 3000');

});

21. **Explain middleware in the context of Express.js.**

Middleware functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application’s request-response cycle.

Example:

app.use((req, res, next) => {

console.log('Request URL:', req.url);

next();

});

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

res.send('Hello, World!');

});

22. **How do you handle routing in Express.js?**

By defining route handlers for different HTTP methods and URL paths.

Example:

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

res.send('GET request to the homepage');

});

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

res.send('POST request to the homepage');

});

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

res.send('PUT request to /user');

});

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

res.send('DELETE request to /user');

});

23. **How do you create RESTful APIs using Express.js?**

By defining routes that correspond to CRUD operations (Create, Read, Update, Delete).

Example:

const express = require('express');

const app = express();

const bodyParser = require('body-parser');

app.use(bodyParser.json());

let items = [];

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

const item = req.body;

items.push(item);

res.status(201).send(item);

});

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

res.send(items);

});

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

const item = items.find(i => i.id ===

parseInt(req.params.id));

if (!item) return res.status(404).send('Item

not found');

res.send(item);

});

app.put('/items/:id', (req, res) => {

const item = items.find(i => i.id ===

parseInt(req.params.id));

if (!item) return res.status(404).send('Item

not found');

item.name = req.body.name;

res.send(item);

});

app.delete('/items/:id', (req, res) => {

const itemIndex = items.findIndex(i => i.id

=== parseInt(req.params.id));

if (itemIndex === -1) return

res.status(404).send('Item not found');

items.splice(itemIndex, 1);

res.status(204).send();

});

app.listen(3000, () => {

console.log('Server running on port 3000');

});

24. **How do you connect Node.js with databases like MongoDB or MySQL?**

Using database drivers or ORMs.

Example with MongoDB and Mongoose:

javascript

const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/myda

tabase', { useNewUrlParser: true,

useUnifiedTopology: true });

const db = mongoose.connection;

db.on('error', console.error.bind(console,

'connection error:'));

db.once('open', function() {

console.log('Connected to MongoDB');

});

Example with MySQL:

const mysql = require('mysql');

const connection = mysql.createConnection({

host: 'localhost',

user: 'root',

password: '',

database: 'mydatabase'

});

connection.connect((err) => {

if (err) throw err;

console.log('Connected to MySQL');

});

25. **What is Mongoose? How do you use it with MongoDB in Node.js?**

Mongoose is an ODM (Object Data Modeling) library for MongoDB and Node.js. It manages relationships between data, provides schema validation, and translates between objects in code and the MongoDB documents.

Example:

const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/myda

tabase', { useNewUrlParser: true,

useUnifiedTopology: true });

const userSchema = new mongoose.Schema({

name: String,

age: Number

});

const User = mongoose.model('User', userSchema);

const user = new User({ name: 'John', age: 30

});

user.save().then(() => console.log('User

saved'));

26. **How do you handle errors in Node.js applications?**

By using try/catch blocks, callbacks with error arguments, Promises, and centralized error handling middleware in Express.

Example:

// Error handling in async/await

async function getData() {

try {

const data = await someAsyncFunction();

console.log(data);

} catch (error) {

console.error('Error:', error);

}

}

27. **What tools and techniques do you use for debugging Node.js applications?**

Tools: node inspect, Chrome DevTools, Visual Studio Code debugger, and libraries like debug.

Example with debug library:

const debug = require('debug')('app');

debug('This is a debug message');

**Security**

28. **What are some common security concerns in Node.js applications?**

Injection attacks (SQL, NoSQL, Command), cross-site scripting (XSS), cross-site request forgery (CSRF), and insecure dependencies.

29. **How do you prevent common security vulnerabilities in Node.js applications?**

Use parameterized queries, sanitize user input, use security headers, validate data, keep dependencies up to date, and use tools like Helmet for setting HTTP headers.

Example:

const helmet = require('helmet');

app.use(helmet());

30. **How do you deploy Node.js applications? What considerations are important for deployment?**

Deploy to platforms like Heroku, AWS, orDigitalOcean. Use process managers like PM2, and configure environments properly.

Example with PM2:

// Install PM2 globally

npm install pm2 -g

// Start application with PM2

pm2 start app.js

31. **What is the event loop in Node.js? How does it work?**

The event loop is a mechanism that allows Node.js to perform non-blocking I/O

operations by offloading operations to the system kernel whenever possible. It

processes asynchronous callbacks and executes them in a single-threaded event-driven loop.

Example:

console.log('Start');

setTimeout(() => {

console.log('Timeout callback');

}, 0);

console.log('End');

// Output:

// Start

// End

// Timeout callback

32. **What are process.nextTick() and setImmediate()? How do they differ?**

process.nextTick() schedules a callback function to be invoked in the next iteration of the event loop, before any I/O operations.

setImmediate() schedules a callback to be executed after I/O events callbacks and before timers.

Example:

console.log('Start');

process.nextTick(() => {

console.log('Next Tick');

});

setImmediate(() => {

console.log('Immediate');

});

console.log('End');

// Output:

// Start

// End

// Next Tick

// Immediate

33. **Explain the concept of clustering in Node.js. How does it improve performance?**

Clustering allows Node.js to create multiple processes that share the same server port. This is useful for taking advantage of multi-core systems to handle more concurrent connections.

Example:

const cluster = require('cluster');

const http = require('http');

const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {

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

cluster.fork();

}

cluster.on('exit', (worker, code, signal) => {

console.log(`Worker ${worker.process.pid}

died`);

});

} else {

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

res.writeHead(200);

res.end('Hello World\n');

}).listen(8000);

}

34. **What are worker threads in Node.js? How do they differ from clustering?**

Worker threads allow the execution of JavaScript code in parallel using multiple

threads. Unlike clustering, worker threads can share memory and are suitable for

CPU-intensive tasks.

Example:

const { Worker, isMainThread, parentPort,

workerData } = require('worker\_threads');

if (isMainThread) {

new Worker(\_\_filename, { workerData: 'Hello,

Worker' });

} else {

console.log(workerData); // 'Hello, Worker'

}

35. **How do you write tests for Node.js applications? What frameworks do you use?**

○ Use testing frameworks like Mocha, Jest, and

Chai to write and run tests.

Example with Mocha and Chai:

javascript

Copy code

const { expect } = require('chai');

const { add } = require('./math');

describe('Math Functions', () => {

it('should add two numbers', () => {

const result = add(2, 3);

expect(result).to.equal(5);

});

});

36. **What is the purpose of mocking in tests? How do you use Sinon for mocking?**

○ Mocking simulates the behavior of complex

or external dependencies in tests, allowing

you to test units of code in isolation.

Example with Sinon:

javascript

Copy code

const sinon = require('sinon');

const { expect } = require('chai');

const userService = require('./userService');

describe('User Service', () => {

it('should call getUser once', () => {

const getUser = sinon.spy(userService,

'getUser');

userService.getUser(1);

expect(getUser.calledOnce).to.be.true;

getUser.restore();

});

});

○

37. **How do you handle asynchronous testing in**

**Node.js?**

○ Use done callback, Promises, or async/await

to handle asynchronous code in tests.

Example with async/await:

javascript

Copy code

const { expect } = require('chai');

const { fetchData } = require('./dataService');

describe('Data Service', () => {

it('should fetch data', async () => {

const data = await fetchData();

expect(data).to.equal('Expected Data');

});

});

○

**Performance Optimization**

38. **What are some common performance optimization**

**techniques in Node.js?**

○ Caching, using streams, clustering, avoiding

synchronous code, optimizing database

queries, and monitoring performance with

tools like New Relic or PM2.

39. **How do you profile and monitor Node.js**

**applications?**

○ Use tools like Node.js built-in profiler,

Clinic.js, New Relic, or PM2 to monitor

performance and identify bottlenecks.

Example with Clinic.js:

sh

Copy code

npm install -g clinic

clinic doctor -- node your-app.js

○

**Miscellaneous**

40. **What is the role of environment variables in Node.js**

**applications?**

○ Environment variables are used to store

configuration and secrets, allowing

applications to be configured differently in

various environments (development, testing,

production).

Example:

javascript

Copy code

// Access environment variable

const port = process.env.PORT || 3000;

app.listen(port, () => {

console.log(`Server running on port ${port}`);

});

○

41. **How do you handle configuration management in**

**Node.js?**

○ Use libraries like dotenv for loading

environment variables from a .env file, and

config for managing configuration settings.

Example with dotenv:

javascript

Copy code

require('dotenv').config();

console.log(process.env.DB\_HOST); // Value from

.env file

○

42. **What are some best practices for structuring**

**Node.js applications?**

○ Use a modular structure, separate business

logic from presentation, use middleware for

cross-cutting concerns, and follow the single

responsibility principle.

Example structure:

go

Copy code

myapp/

├── controllers/

│ └── userController.js

├── models/

│ └── userModel.js

├── routes/

│ └── userRoutes.js

├── services/

│ └── userService.js

├── app.js

└── package.json

○

43. **How do you handle static files in Express.js?**

○ Use the express.static middleware to

serve static files.

Example:

javascript

Copy code

const express = require('express');

const app = express();

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

app.listen(3000, () => {

console.log('Server running on port 3000');

});

○

44. **What is the purpose of middleware in Express.js?**

○ Middleware functions execute during the

request-response cycle, allowing you to

modify the request or response objects, end

the request-response cycle, or call the next

middleware function.

45. **How do you implement authentication and**

**authorization in Node.js applications?**

○ Use libraries like Passport.js for

authentication and implement role-based

access control for authorization.

Example with Passport.js:

javascript

Copy code

const passport = require('passport');

const LocalStrategy =

require('passport-local').Strategy;

passport.use(new LocalStrategy(

(username, password, done) => {

// Verify username and password

User.findOne({ username }, (err, user) => {

if (err) return done(err);

if (!user || user.password !== password)

return done(null, false);

return done(null, user);

});

}

));

app.post('/login',

passport.authenticate('local', {

successRedirect: '/',

failureRedirect: '/login'

}));

○

**Conclusion**

These additional questions cover advanced topics, testing,

performance optimization, and various best practices in Node.js

development. Together with the previous questions, this

comprehensive guide provides a solid foundation for preparing for

Node.js interviews.

4o

continue

ChatGPT

**Advanced Topics in Databases**

51. **Explain the differences between SQL and NoSQL**

**databases.**

○ SQL databases are relational, use structured

query language (SQL) for defining and

manipulating data, and have a predefined

schema. Examples include MySQL,

PostgreSQL, and Oracle.

○ NoSQL databases are non-relational, can

store unstructured data, and have dynamic

schemas. They are designed for distributed

data stores with large-scale data storage

needs. Examples include MongoDB,

Cassandra, and Redis.

○ Example:

SQL Query:

sql

Copy code

SELECT \* FROM users WHERE id = 1;

■

NoSQL Query (MongoDB):

javascript

Copy code

db.users.find({ \_id: 1 });

■

52. **What are the advantages and disadvantages of**

**using a NoSQL database like MongoDB?**

○ Advantages:

■ Flexible schema: Easy to modify

the structure of documents.

■ Scalability: Designed to scale out

by adding more servers.

■ Performance: Can handle large

volumes of unstructured data.

○ Disadvantages:

■ Consistency: May not be strongly

consistent (depends on the

implementation).

■ Complex queries: Limited support

for complex queries and joins

compared to SQL.

■ Data redundancy: Denormalization

may lead to data redundancy.

Example:

javascript

Copy code

// Flexible schema

const user = { name: "John Doe", age: 30,

address: { city: "New York" } };

db.users.insert(user);

○

**Data Modeling in MongoDB**

53. **How do you model one-to-many relationships in**

**MongoDB?**

○ Use embedded documents or references.

Embedded documents are suitable for small,

bounded data, while references are better for

large or frequently updated data.

○ Example:

Embedded documents:

javascript

Copy code

const blogPost = {

title: "Post Title",

content: "Post content...",

comments: [

{ user: "Alice", text: "Great post!" },

{ user: "Bob", text: "Thanks for sharing!" }

]

};

db.posts.insert(blogPost);

■

References:

javascript

Copy code

const post = { title: "Post Title", content:

"Post content..." };

db.posts.insert(post);

const comment = { postId: post.\_id, user:

"Alice", text: "Great post!" };

db.comments.insert(comment);

■

54. **How do you model many-to-many relationships in**

**MongoDB?**

○ Use an array of references to represent

many-to-many relationships.

Example:

javascript

Copy code

const student = { name: "John Doe", courses:

[course1\_id, course2\_id] };

const course = { title: "Math 101", students:

[student1\_id, student2\_id] };

db.students.insert(student);

db.courses.insert(course);

○

**Indexing and Performance in MongoDB**

55. **What is the purpose of indexing in MongoDB?**

○ Indexing improves the performance of

queries by allowing the database to quickly

locate documents that match the query

criteria.

Example:

javascript

Copy code

db.users.createIndex({ name: 1 }); // Create an

index on the 'name' field

○

56. **How do you analyze and optimize query**

**performance in MongoDB?**

○ Use the explain() method to analyze

query execution and identify performance

bottlenecks.

Example:

javascript

Copy code

db.users.find({ age: { $gt: 25 }

}).explain("executionStats");

○

**Transactions in MongoDB**

57. **What are MongoDB transactions? How do you use**

**them?**

○ MongoDB transactions provide ACID

(Atomicity, Consistency, Isolation, Durability)

guarantees, allowing multiple operations to

be executed in an all-or-nothing manner.

Example:

javascript

Copy code

const session = client.startSession();

session.startTransaction();

try {

db.users.updateOne({ \_id: 1 }, { $set: {

balance: 50 } }, { session });

db.accounts.updateOne({ \_id: 2 }, { $set: {

balance: 150 } }, { session });

session.commitTransaction();

} catch (error) {

session.abortTransaction();

} finally {

session.endSession();

}

○

**Replication and Sharding in MongoDB**

58. **What is replication in MongoDB? How do you set**

**up a replica set?**

○ Replication is the process of synchronizing

data across multiple servers to ensure high

availability and data redundancy.

Example setup:

javascript

Copy code

rs.initiate({

\_id: "rs0",

members: [

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

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

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

]

});

○

59. **What is sharding in MongoDB? How do you set up**

**sharding?**

○ Sharding is the process of distributing data

across multiple servers to handle large

datasets and high throughput operations.

Example setup:

javascript

Copy code

sh.addShard("shard1/localhost:27020");

sh.addShard("shard2/localhost:27021");

sh.addShard("shard3/localhost:27022");

sh.enableSharding("myDatabase");

sh.shardCollection("myDatabase.myCollection", {

shardKey: 1 });

○

**Security in MongoDB**

60. **What are some best practices for securing**

**MongoDB installations?**

○ Enable authentication, use strong passwords,

enforce role-based access control, enable

SSL/TLS for encrypted connections, and

regularly update MongoDB to the latest

version.

Example:

javascript

Copy code

// Enable authentication in the configuration

file

security:

authorization: "enabled"

○

61. **How do you prevent NoSQL injection attacks in**

**MongoDB?**

○ Validate and sanitize user inputs, use

parameterized queries, and limit the types of

operations that users can perform.

Example:

javascript

Copy code

const userId = sanitize(req.body.userId);

db.users.findOne({ \_id: userId });

○

**Integration and Tools**

62. **How do you integrate MongoDB with Node.js**

**applications?**

○ Use the official MongoDB Node.js driver to

connect and interact with MongoDB.

Example:

javascript

Copy code

const { MongoClient } = require('mongodb');

const uri = "mongodb://localhost:27017";

const client = new MongoClient(uri);

async function run() {

try {

await client.connect();

const database = client.db('test');

const collection =

database.collection('users');

const user = await collection.findOne({

name: "John Doe" });

console.log(user);

} finally {

await client.close();

}

}

run().catch(console.dir);

○

63. **What are some useful MongoDB management tools**

**and libraries?**

○ Tools like MongoDB Atlas for cloud-based

management, Robo 3T for GUI-based

interaction, and libraries like Mongoose for

object data modeling in Node.js.

Example with Mongoose:

javascript

Copy code

const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost/test', {

useNewUrlParser: true, useUnifiedTopology: true

});

const userSchema = new mongoose.Schema({

name: String,

age: Number,

email: String

});

const User = mongoose.model('User', userSchema);

const newUser = new User({ name: 'John Doe',

age: 30, email: 'john.doe@example.com' });

newUser.save().then(() => console.log('User

saved!'));

○

**Testing MongoDB**

64. **How do you write tests for MongoDB interactions in**

**Node.js applications?**

○ Use testing frameworks like Mocha and Chai,

and mock the database using tools like

mongodb-memory-server for in-memory

MongoDB instances.

Example:

javascript

Copy code

const { MongoMemoryServer } =

require('mongodb-memory-server');

const mongoose = require('mongoose');

const { expect } = require('chai');

const User = require('./userModel'); // Assuming

a Mongoose model

let mongoServer;

before(async () => {

mongoServer = new MongoMemoryServer();

const uri = await mongoServer.getUri();

await mongoose.connect(uri, { useNewUrlParser:

true, useUnifiedTopology: true });

});

after(async () => {

await mongoose.disconnect();

await mongoServer.stop();

});

describe('User Model', () => {

it('should create a new user', async () => {

const user = new User({ name: 'John Doe',

age: 30, email: 'john.doe@example.com' });

const savedUser = await user.save();

expect(savedUser.name).to.equal('John Doe');

});

});

○

This comprehensive list of questions and examples covers

advanced topics in both backend development with Node.js and

database management with MongoDB, providing a solid

foundation for preparing for technical interviews in these areas.

4o

continue

ChatGPT

**Advanced MongoDB Features**

65. **What is the aggregation framework in MongoDB?**

**How does it work?**

○ The aggregation framework processes data

records and returns computed results. It

operates on the concept of data pipelines,

where documents are processed in stages.

Example:

javascript

Copy code

db.orders.aggregate([

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

{ $group: { \_id: "$cust\_id", total: { $sum:

"$amount" } } },

{ $sort: { total: -1 } }

]);

○

66. **How do you perform text search in MongoDB?**

○ MongoDB supports text search through text

indexes. You can create text indexes on fields

containing string data.

Example:

javascript

Copy code

db.articles.createIndex({ title: "text",

content: "text" });

db.articles.find({ $text: { $search: "mongodb

tutorial" } });

○

67. **What are MongoDB change streams? How do you**

**use them?**

○ Change streams allow applications to access

real-time data changes without the

complexity and risk of tailing the oplog. They

provide a way to listen to changes on a

collection.

Example:

javascript

Copy code

const changeStream =

db.collection('orders').watch();

changeStream.on('change', (change) => {

console.log('Change detected:', change);

});

○

68. **Explain the Map-Reduce function in MongoDB.**

**When would you use it?**

○ Map-Reduce is a data processing paradigm

for condensing large volumes of data into

useful aggregated results. It involves a map

function to process each document and a

reduce function to combine the results.

Example:

javascript

Copy code

db.orders.mapReduce(

function() { emit(this.cust\_id, this.amount);

},

function(key, values) { return

Array.sum(values); },

{ out: "order\_totals" }

);

db.order\_totals.find();

○

**MongoDB Administration**

69. **How do you monitor MongoDB performance?**

○ Use tools like MongoDB Cloud Manager, Ops

Manager, or open-source monitoring tools

like mongostat, mongotop, and the

MongoDB Profiler.

Example:

shell

Copy code

mongostat --host localhost --port 27017

○

70. **How do you perform a backup and restore of a**

**MongoDB database?**

○ Use the mongodump and mongorestore

tools to back up and restore MongoDB

databases.

Example:

shell

Copy code

mongodump --out /data/backup/

mongorestore --dir /data/backup/

○

**MongoDB Atlas**

71. **What is MongoDB Atlas? How do you deploy a**

**database using Atlas?**

○ MongoDB Atlas is a fully-managed cloud

database service. It provides automated

provisioning, patching, backups, and

monitoring.

○ Example:

■ Sign up for MongoDB Atlas.

■ Create a new cluster.

■ Configure security settings (IP

whitelist, user roles).

■ Connect to the cluster using the

provided connection string.

javascript

Copy code

const { MongoClient } = require('mongodb');

const uri = "your\_atlas\_connection\_string";

const client = new MongoClient(uri, {

useNewUrlParser: true, useUnifiedTopology: true

});

async function run() {

try {

await client.connect();

const database = client.db('test');

const collection =

database.collection('users');

const user = await collection.findOne({

name: "John Doe" });

console.log(user);

} finally {

await client.close();

}

}

run().catch(console.dir);

○

72. **How do you scale a MongoDB cluster on Atlas?**

○ Scaling a MongoDB cluster on Atlas involves

upgrading instance sizes, adding more

shards for horizontal scaling, or enabling

auto-scaling features.

○ Example:

■ Navigate to the Cluster page on

Atlas.

■ Click on the "Scale" button.

■ Choose the new cluster size and

configuration.

■ Apply changes to scale the cluster.

**MongoDB Best Practices**

73. **What are some best practices for schema design in**

**MongoDB?**

○ Design schema according to application

requirements and query patterns.

○ Prefer embedding for small sub-documents

and referencing for large or frequently

updated documents.

○ Avoid large documents (16MB limit) and

deeply nested documents.

○ Use appropriate data types for fields.

Example:

javascript

Copy code

const userSchema = {

name: String,

email: String,

profile: {

age: Number,

address: String

}

};

○

74. **How do you handle large datasets in MongoDB?**

○ Use sharding to distribute large datasets

across multiple servers.

○ Optimize queries and indexes to improve

performance.

○ Use the aggregation framework for complex

data processing.

Example:

javascript

Copy code

sh.enableSharding("myDatabase");

sh.shardCollection("myDatabase.myCollection", {

shardKey: 1 });

○

**Query Optimization**

75. **How do you optimize MongoDB queries for**

**performance?**

○ Use indexes to speed up query performance.

○ Analyze query performance using

explain().

○ Avoid using unindexed fields in queries.

○ Optimize schema design to align with query

patterns.

Example:

javascript

Copy code

db.collection.createIndex({ field1: 1, field2:

-1 });

db.collection.find({ field1: value1

}).explain("executionStats");

○

76. **What is the $geoNear aggregation stage in**

**MongoDB?**

○ The $geoNear aggregation stage calculates

distances from a point and filters documents

within a specified range.

Example:

javascript

Copy code

db.places.aggregate([

{

$geoNear: {

near: { type: "Point", coordinates: [

-73.9667, 40.78 ] },

distanceField: "dist.calculated",

maxDistance: 2000,

query: { type: "restaurant" },

spherical: true

}

}

]);

○

**MongoDB Transactions**

77. **How do you implement multi-document**

**transactions in MongoDB?**

○ Use sessions to start a transaction and

include multiple operations in the transaction

scope.

Example:

javascript

Copy code

const session = client.startSession();

session.startTransaction();

try {

db.users.updateOne({ \_id: 1 }, { $set: {

balance: 50 } }, { session });

db.accounts.updateOne({ \_id: 2 }, { $set: {

balance: 150 } }, { session });

session.commitTransaction();

} catch (error) {

session.abortTransaction();

} finally {

session.endSession();

}

○

**MongoDB Performance Monitoring**

78. **How do you use the MongoDB profiler?**

○ The MongoDB profiler collects fine-grained

data about MongoDB operations, such as

CRUD operations and queries.

Example:

javascript

Copy code

db.setProfilingLevel(2);

db.system.profile.find({ millis: { $gt: 100 }

}).sort({ ts: -1 });

○

**Advanced Indexing Techniques**

79. **What are partial indexes in MongoDB?**

○ Partial indexes index only a subset of

documents in a collection based on a filter

expression.

Example:

javascript

Copy code

db.users.createIndex({ age: 1 }, {

partialFilterExpression: { age: { $gt: 21 } }

});

○

80. **What are compound indexes in MongoDB?**

○ Compound indexes index multiple fields

within a document and support more efficient

queries on those fields.

Example:

javascript

Copy code

db.orders.createIndex({ status: 1, date: -1 });

○

This comprehensive list of advanced MongoDB features and

techniques covers essential aspects of working with MongoDB in

real-world applications, providing a solid foundation for preparing

for technical interviews and improving your proficiency with

MongoDB.

4o

continue

ChatGPT

**MongoDB Aggregation and Querying**

81. **How do you use the $project stage in MongoDB**

**aggregation?**

○ The $project stage is used to include,

exclude, or add new fields to documents

being passed through the aggregation

pipeline.

Example:

javascript

Copy code

db.orders.aggregate([

{ $project: { \_id: 0, item: 1, total: {

$multiply: ["$price", "$quantity"] } } }

]);

○

82. **What is the $unwind stage in MongoDB**

**aggregation?**

○ The $unwind stage deconstructs an array

field from the input documents to output a

document for each element.

Example:

javascript

Copy code

db.orders.aggregate([

{ $unwind: "$items" },

{ $project: { \_id: 0, order\_id: 1, item:

"$items.name", quantity: "$items.qty" } }

]);

○

83. **Explain the $group stage in MongoDB aggregation.**

○ The $group stage groups input documents

by a specified identifier expression and

applies the accumulator expressions to each

group.

Example:

javascript

Copy code

db.sales.aggregate([

{ $group: { \_id: "$item", totalSales: { $sum:

"$amount" }, avgQuantity: { $avg: "$quantity" }

} }

]);

○

84. **How do you use the $match stage in MongoDB**

**aggregation?**

○ The $match stage filters documents to pass

only those documents that match the

specified condition(s) to the next pipeline

stage.

Example:

javascript

Copy code

db.orders.aggregate([

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

{ $group: { \_id: "$cust\_id", total: { $sum:

"$amount" } } }

]);

○

**MongoDB Geospatial Queries**

85. **How do you perform geospatial queries in**

**MongoDB?**

○ MongoDB supports geospatial indexes and

queries, allowing for operations like finding

documents near a specific location.

Example:

javascript

Copy code

db.places.createIndex({ location: "2dsphere" });

db.places.find({

location: {

$near: {

$geometry: { type: "Point", coordinates: [

-73.9667, 40.78 ] },

$maxDistance: 1000

}

}

});

○

86. **Explain the use of the $geoWithin operator in**

**MongoDB.**

○ The $geoWithin operator selects

documents with geospatial data within a

specified geometry.

Example:

javascript

Copy code

db.places.find({

location: {

$geoWithin: {

$geometry: { type: "Polygon", coordinates:

[ [ [ -73.97, 40.77 ], [ -73.97, 40.78 ], [

-73.96, 40.78 ], [ -73.96, 40.77 ], [ -73.97,

40.77 ] ] ] }

}

}

});

○

**Advanced MongoDB Indexing**

87. **What are wildcard indexes in MongoDB?**

○ Wildcard indexes index fields with dynamic or

unknown keys and provide a way to index all

fields within documents.

Example:

javascript

Copy code

db.collection.createIndex({ "$\*\*": 1 });

○

88. **Explain how to use hashed indexes in MongoDB.**

○ Hashed indexes use hashed values of the

indexed field, providing a uniform distribution

of indexes.

Example:

javascript

Copy code

db.collection.createIndex({ user\_id: "hashed"

});

○

**MongoDB Performance Tuning**

89. **What are some techniques to optimize MongoDB**

**performance?**

○ Optimize schema design.

○ Use proper indexing.

○ Perform query optimization and analysis

using explain().

○ Use aggregation pipelines effectively.

Example:

javascript

Copy code

db.orders.find({ status: "A"

}).explain("executionStats");

○

90. **How do you use the $sample stage in MongoDB**

**aggregation?**

○ The $sample stage randomly selects a

specified number of documents from its input.

Example:

javascript

Copy code

db.orders.aggregate([

{ $sample: { size: 5 } }

]);

○

**MongoDB Data Validation**

91. **How do you enforce data validation in MongoDB?**

○ Use schema validation rules to enforce data

integrity in collections.

Example:

javascript

Copy code

db.createCollection("users", {

validator: {

$jsonSchema: {

bsonType: "object",

required: [ "name", "email" ],

properties: {

name: { bsonType: "string" },

email: { bsonType: "string", pattern:

"^.+@.+$" }

}

}

}

});

○

92. **Explain how to use custom validation in MongoDB.**

○ Custom validation involves using JavaScript

functions to enforce complex validation rules.

Example:

javascript

Copy code

db.runCommand({

collMod: "users",

validator: {

$jsonSchema: {

bsonType: "object",

required: [ "name", "email" ],

properties: {

name: { bsonType: "string" },

email: { bsonType: "string", pattern:

"^.+@.+$" }

}

}

},

validationAction: "warn"

});

○

**MongoDB Data Migration**

93. **How do you handle data migrations in MongoDB?**

○ Use tools like mongodump and

mongorestore, or use the MongoDB Atlas

Data Lake.

○ Plan schema changes carefully and use

scripts or tools for data migration.

Example:

shell

Copy code

mongodump --uri mongodb://localhost:27017/mydb

--out /data/backup/

mongorestore --uri mongodb://newhost:27017/mydb

--dir /data/backup/

○

**MongoDB Query Operators**

94. **Explain the $elemMatch operator in MongoDB.**

○ The $elemMatch operator matches

documents that contain an array field with at

least one element that matches all the

specified query criteria.

Example:

javascript

Copy code

db.inventory.find({

items: { $elemMatch: { qty: { $gt: 10, $lt: 20

} } }

});

○

95. **How do you use the $regex operator in MongoDB?**

○ The $regex operator provides regular

expression capabilities for pattern matching

strings in queries.

Example:

javascript

Copy code

db.users.find({ name: { $regex: /^A/, $options:

'i' } });

○

**MongoDB Schema Design Patterns**

96. **What are some common schema design patterns in**

**MongoDB?**

○ **Single View Pattern:** Aggregating data from

multiple sources into a single document.

○ **Tree Pattern:** Using parent references, child

references, or materialized paths to represent

hierarchical data.

○ **Subset Pattern:** Storing frequently accessed

data in the same document to reduce the

number of reads.

Example:

javascript

Copy code

const userSchema = {

name: String,

email: String,

profile: {

age: Number,

address: String

},

orders: [{ order\_id: String, amount: Number }]

};

○

**MongoDB GridFS**

97. **What is GridFS in MongoDB? How do you use it?**

○ GridFS is a specification for storing and

retrieving large files (larger than the

BSON-document size limit of 16 MB).

Example:

javascript

Copy code

const GridFSBucket =

require('mongodb').GridFSBucket;

const bucket = new GridFSBucket(db, {

bucketName: 'files' });

const uploadStream =

bucket.openUploadStream('myfile.txt');

fs.createReadStream('path/to/myfile.txt').pipe(u

ploadStream);

const downloadStream =

bucket.openDownloadStreamByName('myfile.txt');

downloadStream.pipe(fs.createWriteStream('path/t

o/myfile.txt'));

○

**MongoDB Best Practices**

98. **What are some best practices for working with**

**MongoDB?**

○ Design schemas according to application

requirements and query patterns.

○ Use indexes to optimize query performance.

○ Avoid large or deeply nested documents.

○ Regularly monitor and optimize database

performance.

Example:

javascript

Copy code

db.collection.createIndex({ field: 1 });

db.collection.find({ field: value

}).explain("executionStats");

○

99. **How do you ensure data consistency and integrity**

**in MongoDB?**

○ Use transactions for multi-document

operations.

○ Implement schema validation.

○ Regularly back up and test data recovery

processes.

Example:

javascript

Copy code

const session = client.startSession();

session.startTransaction();

try {

db.collection1.updateOne({ \_id: 1 }, { $set: {

field: value } }, { session });

db.collection2.updateOne({ \_id: 1 }, { $set: {

field: value } }, { session });

session.commitTransaction();

} catch (error) {

session.abortTransaction();

} finally {

session.endSession();

}

○

**MongoDB Future Trends**

100. **What are some future trends and developments in**

**MongoDB?** - Continued improvement in cloud-based

database services like MongoDB Atlas. - Enhanced

support for real-time analytics and processing with

features like change streams. - Advancements in

performance optimization and indexing strategies. -

Integration with other technologies and platforms for

better data management and utilization.

This comprehensive set of advanced MongoDB features and

techniques provides a thorough foundation for understanding and

leveraging MongoDB's capabilities in various scenarios, from

development to administration and optimization.

**MONGO DB**

**No sql** is a type of data base designed to store, manage and retrieve data without structural format.

It allows for flexible schemas, scalability, and high performance, making it suitable for large-scale applications and real time applications.

Handle key-pair values, graphs and wide columns.

**Mongo DB** is a no sql DB that uses a flexible, document-oriented data model to store data in JSON like BSON format.

It allows dynamic schemas, large volume of data across distributed system.

**JSON**: JavaScript object notation is a light weight data interchange.

It is a simple format to represent structured data object, array and key-pair value.

**BSON**: Binary encoded serialization format that extends JSON like adding data type likes integer, and dates making it more efficient for storing and retrieval.

It is primary data format used by mongo DB for storing documents and making procedure calls.

**INTEGRATION OF MONGODB**

npm install mongoose

**const** *mongoose* = require(**'mongoose'**);

*mongoose*.connect(**'mongodb://localhost:27017/mern-app'**)

.then(()**=>**{

*console*.log(**'DB connected'**)

})

.catch((*err*)**=>**{

*console*.log(*err*);

})

**Connecting to MongoDB Server**

Step 1: Open command prompt(cmd) in your system

Step 2: Type 'mongosh' then hit enter key

**Show database**

show dbs: It will show all the available databases

**Command to connect database**

#Syntax: use [database name]: It will pick you which data base need to work on

|  |  |
| --- | --- |
| How to delete collection:   |  | | --- | | db.collectionname.drop() ex: db.orders.drop() | |

**CRUD Operation**

1. Create
   * insertOne(data,options)
   * insertMany(data,options)

**Creating Collection & Document**

#Syntax: db.collectionName.insertOne({document})

#Example db.products.insertOne({name:'Mac Book Pro',price:120000,brand:'Apple'})

Under the database need to create a collection in that we have to inserted one document.

Example: db.products.insertMany([{name:'Iphone14',price:75000,brand:'Apple'},{name:'Samsung Galaxy A50',price:54000,brand:'Samsung'}])

Under the database need to create a collection in that we have to insert more than one document.

1. Read
   * find(filter,options)
   * findOne(filter,options) #first matching document

**To get all the data from the collection**

#Syntax: db.collectionName.find(): It will show us the document available in that particular collection(1st set)

db.customers.find().toArray():All data

db.customers.find().forEach((customersData)=> {printjson(customersData)})

db.products.find().forEach((*customersData*)=> {*printjson*(*customersData*)})

db.products.find({},{name:1,price:1}) :it will list only name and price along with id

db.products.find({},{\_id:0,name:1,price:1}) :it will list only name and price.

#Syntax: db.collectionName.find({particular document}): It will show us current document available in that particular db

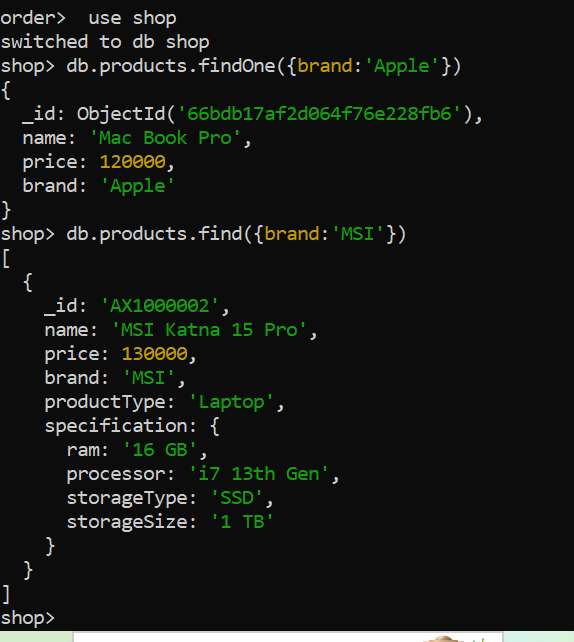
#Syntax: db.collectionName.findOne({particular document}): It will show us first one of the current document available in that particular db

Example:

db.products.find({brand:'Apple'}) :

Example:

db.products.findOne({brand:'Apple'})



1. Update
   * updateOne(filter,data,options)
   * updateMany(filter,data,options)
   * replaceOne(filter,data,options)

**Update document in collection**

#Syntax: Db.collectionName.updateOne({},{}): filter and what to set

**Example:**

db.products.updateOne({name:'Mac Book Pro'},{$set:{productType:'Laptop'}})

**Example:**

db.products.updateOne({name:'Samsung Galaxy s24 Ultra'},{$set:{price:93000}})

**Example:**

db.products.updateOne({name:'Bravia 43 inch 4k'},{$set:{price:45000}})

**Example:**

db.products.updateOne({name:'Iphone 15'},{$set:{brand:'Apple',productType:'Mobile'}})

**Example:**

db.products.updateMany({\_id:{$in:[ObjectId('66695c4aa52b865509cdcdf8'),ObjectId('66695c8ba52b865509cdcdfd')]}},{$set:{productType:"Mobile"}})

**Example update many**

db.products.updateMany({\_id:{$in:[ObjectId('66695c4aa52b865509cdcdf8'),ObjectId('66695c8ba52b865509cdcdfd')]}},{$set:{productType:"Mobile"}})

**Replacing of entire document:**

db.products.replaceOne({\_id:ObjectId('66695c8ba52b865509cdcdfc')},{name:"Iphone 14 Pro Max",price:115000,brand:'Apple',productType:'Mobile'})

1. Delete
   * deleteOne(filter,options)
   * deleteMany(filter,options)

## Delete document from collection

# the following command will delete the 1st document with matched filter

db.products.deleteOne({brand:'Apple'})

# the following command will delete document many at same time

db.products.deleteMany({productType:'Laptop'}) documents matches the filter will delete from collections

db.products.deleteMany({})--- all documents will delete from collections

**deleting collection:**

db.products.drop()---collection will delete

**Deleting Database:**

db.dropDatabase()

**What is Aggregation in Mongo DB?**

It groups values from multiple documents and returns computed results.

It can perform a variety of operations on grouped data to return a result.

The aggregation pipeline.

The map reduce function

Single purpose aggregation methods and commands.

**What is shardding in Mongo DB?**

It’s a method to store data across multiple machines.

Supports deployment with large data.

**What is Mongoose in Mongo DB?**

Is an ODM it allows objects or models with strongly typed schema.

Schema based modelling system.

Built-in type casting, query building and business logic.

**What is Replication in Mongo DB?**

It is the process of synchronizing data across multiple servers.

It provides redundancy and increases data availability.

Replication protects a database from the loss of single server.

It also you to recover from hardware failures and service interruption.

**What is MongoDB?**

**What is NoSQL?**

**What is the difference between NoSQL & RDBMS?**

**When to use RDBMS and when to use NoSQL DB in your applications?**

**What are Documents and Collections in NoSQL?**

**What are the advantages of NoSQL over RDBMS/SQL databases?**

**What are the dis-advantages of NoSQL as comparison to RDBMS databases?**

**What is the difference between MongoDB Server and MongoDB Database?**

**How to install and run MongoDB server? What is MongoDB compass?**

**How to create database and a collection in MongoDB server?**

**What are CRUD operations in MongoDB?**

**How to Connect to MongoDB server from Node.js application?**

**How to create/ insert a record in MongoDB via Node.js?**

**How to read/ select a record in MongoDB via Node.js?**

**How to update a record in MongoDB via Node.js?**

**How to delete a record in MongoDB via Node.js?**

**How to insert, select, update & delete multiple records in MongoDB via Node.js?**

**What are Query Operators in MongoDB?**

**Compare Query Operators in SQL & MongoDB?**

**How to implement comparison operators?**

**What are in, nin, exists, and, or, regex, not query operators in MongoDB?**

**What is projection in MongoDB? How to implement it?**

**How to remove \_id field from result in MongoDB?**

**How do you perform sorting in ascending & descending order in MongoDB?**

**What are Indexes in MongoDB? How indexing make data retrieval faster?**

**What are indexes? How indexing make data retrieval faster?**

**How to create indexes in MongoDB?**

**On which filed you will create an index - projection field or filter field?**

**What happens if I create an index on all fields of the collection?**

**What is the difference between Single Field Indexes & Compound Indexes?**

**What is Mongoose? What are the advantages of using it?**

**How to install Mongoose in Node.js?**

**What is the role of Schema in Mongoose? How to define it?**

**How to perform database operation using Mongoose with Node.js?**

**How Mongoose will do data validation for Node applications?**

**How to perform CRUD operations using Mongoose in Node.js application?**

**How to define relationships between collections using Mongoose?**

**How to create a Mongoose model from a schema?**

**How Mongoose provide middleware support? What are Pre & Post Hooks?**

**EXPRESS JS**

It is framework for NODE JS

npm install express.-🡪 **package-lock.json will create**

In package.json-🡪 **under dependencies express will be available.**

import *express* from **"express"**; **so that we can use express JS in node.JS**

**const** *app* =  express();-🡪**express() module calling as function. (Instance of express)**

(OR)

const express = require('express');

const app = express();