NODE-JS [2025]

|  |
| --- |
|  |

# : Introduction:

* **What is NodeJS?**  
  NodeJS is a JavaScript runtime built on Chrome’s V8 engine, used to run JavaScript outside the browser.
* **Why NodeJS?**  
  It is fast, scalable, event-driven, and great for real-time applications like chats and streaming.
* **Features of NodeJS?**  
  non-blocking I/O, single-threaded but concurrent, cross-platform support, and a huge npm ecosystem.
* **Installation of NodeJS?**  
  Download from nodejs.org and install; verify with node -v and npm -v.
* **How to run NodeJS?**  
  Create a .js file and run it with node filename.js, or use node command for REPL mode.
* **What is OS?**  
  OS (Operating System) is system software that manages hardware and software resources. Examples: Windows, Linux, macOS.
* **What is npm?**  
  npm (Node Package Manager) is a tool to install, update, and manage NodeJS packages.
* **What is npx?**  
  npx is a package runner that executes Node packages without installing them globally.
* **What is process in NodeJS?**

process is a global object in NodeJS that provides information and control about the current Node.js runtime and program execution.

* **Top 15 Properties of process**
* **process.pid** → Returns the process ID of the Node.js process.
* **process.ppid** → Returns the parent process ID.
* **process.version** → Shows Node.js version.
* **process.versions** → Returns versions of Node and its dependencies (V8, OpenSSL).
* **process.platform** → Shows the operating system platform (e.g., win32, linux).
* **process.arch** → Shows the processor architecture (e.g., x64, arm).
* **process.title** → Current process name/title.
* **process.cwd()** → Returns the current working directory.
* **process.env** → Returns environment variables object.
* **process.uptime()** → Returns how long the process has been running.
* **process.memoryUsage()** → Returns memory usage details.
* **process.hrtime()** → High-resolution real time in [seconds, nanoseconds].
* **process.execPath** → Absolute path of the Node.js executable.
* **process.argv** → Array of command-line arguments.
* **process.exitCode** → Exit code to be used when the process exits.
* **Top 15 Methods of process**
* **process.exit([code])** → Ends the process with exit code.
* **process.abort()** → Immediately aborts the process and generates a core dump.
* **process.kill(pid, [signal])** → Sends a signal to a process by PID.
* **process.chdir(directory)** → Changes the current working directory.
* **process.cwd()** → Gets the current working directory.
* **process.hrtime([time])** → Measures precise execution time.
* **process.memoryUsage()** → Returns memory usage info.
* **process.uptime()** → Returns process uptime in seconds.
* **process.nextTick(callback)** → Executes callback after current operation completes.
* **process.on(event, callback)** → Listens for process events like exit, uncaught Exception.
* **process.once(event, callback)** → Event listener that runs only once.
* **process.removeListener(event, callback)** → Removes a previously registered listener.
* **process.setUncaughtExceptionCaptureCallback(fun)** → Handles uncaught exceptions.
* **process.getuid() / process.setuid(id)** → Gets or sets user identity (Unix only).
* **process.getgid() / process.setgid(id)** → Gets or sets group identity (Unix only).

# : NodeJS Fundamentals (FS Modules):

* **What is Module Wrapper in Node.js?**

In Node.js, every file (module) is wrapped inside a **function wrapper** before execution.  
This is called the **Module Wrapper**, and it provides scope isolation so variables inside one module do not leak into others.

* **What are modules?**  
  Modules in NodeJS are reusable blocks of code that can be imported and exported across files.
* **Types of modules?**  
  There are four types of modules in NodeJS.
* **Core Modules**  
  These are built-in modules provided by NodeJS (e.g., fs, http, path, os).  
  They can be used directly without installation using require () or import.
* **2. Local Modules**  
  These are custom/user-defined modules created within your project.  
  They are usually exported with module.exports and imported with require('./filename').
* **3. Third-party Modules**  
  These are external modules installed via **npm** (e.g., express, mongoose).  
  They extend functionality and need installation using npm install.
* **4. Global Modules**  
  These are modules installed globally using npm install -g.  
  They can be accessed from any project without needing local installation (e.g., nodemon, typescript).
* **CommonJS vs ES6?**  
  CommonJS uses require () and module.exports, while ES6 uses import and export.  
  CommonJS loads synchronously, ES6 supports async and is standard in modern JavaScript.
* **Fs modules?**  
  fs (File System) module is a built-in NodeJS module used to work with files and directories.
* **FS Modules Top 20 methods**
* **fs.readFile(path, callback)** – Reads file contents asynchronously.
* **fs.readFileSync(path)** – Reads file contents synchronously.
* **fs.writeFile(path, data, callback)** – Writes data to a file asynchronously, creating or overwriting it.
* **fs.writeFileSync(path, data)** – Writes data to a file synchronously, creating or overwriting it.
* **fs.appendFile(path, data, callback)** – Adds data to the end of a file asynchronously.
* **fs.appendFileSync(path, data)** – Adds data to the end of a file synchronously.
* **fs.open(path, flags, callback)** – Opens a file and returns a file descriptor asynchronously.
* **fs.close(fd, callback)** – Closes an open file descriptor asynchronously.
* **fs.rename(oldPath, newPath, callback)** – Renames or moves a file asynchronously.
* **fs.unlink(path, callback)** – Deletes a file asynchronously.
* **fs.stat(path, callback)** – Retrieves status (like size and timestamps) of a file or directory asynchronously.
* **fs.lstat(path, callback)** – Retrieves status of a symbolic link, not its target.
* **fs.readdir(path, callback)** – Reads contents (files/directories) of a directory asynchronously.
* **fs.mkdir(path, callback)** – Creates a new directory asynchronously.
* **fs.rmdir(path, callback)** – Removes an empty directory asynchronously.
* **fs.copyFile(src, dest, callback)** – Copies a file from source to destination asynchronously.
* **fs.truncate(path, len, callback)** – Resizes a file to the specified length asynchronously.
* **fs.watch(path, listener)** – Watches for changes in a file or directory.
* **fs.watchFile(path, listener)** – Watches a file for changes using polling.
* **fs.unwatchFile(path)** – Stops watching a file set by watchFile.
* **fs.createReadStream(path)** – Creates a readable stream for reading file data.
* **fs.createWriteStream(path)** – Creates a writable stream for writing file data.
* **fs.glob(pattern, callback)** – Asynchronously finds file paths matching a glob pattern (new in v22).
* **fs.globSync(pattern)** – Synchronously finds file paths matching a glob pattern (new in v22).

# : Number System:

* **What is Decimal (Base 10):**
* The standard number system used in daily life.
* Digits: **0–9**.
* Each digit position represents a power of 10.
* Example: 357 = (3 × 10²) + (5 × 10¹) + (7 × 10⁰).
* **What is Octal (Base 8):**
* Digits: **0–7**.
* Each digit position represents a power of 8.
* Example: 347 (octal) = (3 × 8²) + (4 × 8¹) + (7 × 8⁰) = 231 (decimal).
* Used in older computing systems (rare today).
* **Hexadecimal (Base 16):**
* Digits: **0–9 and A–F** (A=10, B=11, …, F=15).
* Very compact way to represent binary (1 hex digit = 4 bits).
* Example: 2F (hex) = (2 × 16¹) + (15 × 16⁰) = 47 (decimal).
* Widely used in programming (e.g., memory addresses, colour codes).
* **Binary (Base 2):**
* Digits: **0 and 1 only**.
* The fundamental number system of computers (1 = ON, 0 = OFF).
* Example: 1011 (binary) = (1 × 2³) + (0 × 2²) + (1 × 2¹) + (1 × 2⁰) = 11 (decimal).
* **Digital Data Units**
* **Bit (b):** Smallest unit of data (0 or 1).
* **Nibble:** 4 bits.
* **Byte (B):** 8 bits (basic storage unit).
* **Kilobyte (KB):** 1 KB = 1024 bytes.
* **Megabyte (MB):** 1 MB = 1024 KB.
* **Gigabyte (GB):** 1 GB = 1024 MB.
* **Terabyte (TB):** 1 TB = 1024 GB.
* Then **Petabyte (PB), Exabyte (EB), Zettabyte (ZB), Yottabyte (YB)**.
* **Character Encoding:**
* **UTF-8:**
* Variable-length encoding: **1–4 bytes per character**.
* Backward compatible with ASCII (first 128 characters = 1 byte).
* Efficient for English (most characters = 1 byte).
* Most common encoding used on the web.
* **UTF-16**
* Uses **2 or 4 bytes per character**.
* Most common characters fit in 2 bytes.
* Less efficient than UTF-8 for English text.
* Used in Windows, Java, and some APIs.
* **UTF-32**
* Fixed length: **4 bytes per character**.
* Direct one-to-one mapping with Unicode code points.
* Very simple to process, but not space-efficient.
* **BOM (Byte Order Mark)**
* A special marker at the beginning of a file.
* Indicates **text encoding** (UTF-8, UTF-16, etc.) and **byte order**.
* Example: UTF-8 BOM = EF BB BF.
* **Endianness**
* Refers to how multi-byte values are stored in memory:
  + **Big Endian:** Most significant byte stored first.
    - Example: 0x12345678 → 12 34 56 78.
  + **Little Endian:** Least significant byte stored first.
    - Example: 0x12345678 → 78 56 34 12.
* Intel CPUs use **Little Endian**.

# : Array Buffer (JS) and Buffer (NodeJS):

* **What is ArrayBuffer in JS?**

An ArrayBuffer is a fixed-length block of raw binary data in memory.

It does not have a format (no numbers, strings, etc.) → you need a **view** (like TypedArray or DataView) to read/write meaningful values.

Commonly used when working with binary files, WebSockets, or low-level networking in browsers.

Example:

let buffer = new ArrayBuffer(8); // 8 bytes of memory

console.log(buffer.byteLength); // 8

* **Reading and Writing in ArrayBuffer?**

You cannot directly read/write in ArrayBuffer. You need **views** like Int8Array, Uint8Array, Float32Array, or DataView.

Example (with DataView):

let buffer = new ArrayBuffer(4);

let view = new DataView(buffer);

* Write

view.setInt16(0, 500); // store at byte 0

* Read

console.log(view.getInt16(0)); // 500

* **Typed Array in JS**

Typed Arrays are array-like objects that represent an array of specific numeric types (e.g., Int8Array, Uint16Array, Float32Array).

They provide a structured way to interpret an ArrayBuffer.

Each type has fixed size (1 byte, 2 bytes, 4 bytes, etc.).

Example:

let buffer = new ArrayBuffer(8);

let int32View = new Int32Array(buffer);

int32View[0] = 42; // write

console.log(int32View[0]); // read → 42

* **Reading & Writing in TypedArray**

Directly index like a normal array. Faster and simpler than DataView.

Example:

let floatView = new Float32Array(4);

floatView[0] = 3.14;

console.log(floatView[0]); // 3.14

* **Node.js Concepts**
* **What is Buffer in Node.js?**

A Buffer in Node.js is a global object used to handle **binary data** directly (outside V8’s normal heap). It is similar to ArrayBuffer in JS but designed specifically for **I/O operations** (files, TCP streams, etc.).

Buffers are instances of Uint8Array with extra methods.

Example:

const buf = Buffer.from("Hello");

console.log(buf); // <Buffer 48 65 6c 6c 6f>

console.log(buf.toString()); // "Hello"

* **Buffer.alloc (size) method?**

Creates a new buffer of given size **initialized with zeros**. safe (clears memory) but slower.

* **Buffer allocUnsafe (size) method?**

Creates a new buffer but does **not initialize memory**.

Faster, but may contain **old/stale data** until overwritten.

* **Buffer.from() method:**

Creates a new buffer from an existing string, array, ArrayBuffer, TypedArray, or another Buffer, converting the data into raw binary form

* **Buffer Pool:**

Node.js internally maintains a **buffer pool** (pre-allocated memory chunk).

Small buffer requests are served from this pool instead of allocating new memory each time.

Improves performance and reduces memory fragmentation.

Example:

When you call Buffer.alloc(5) multiple times, Node may reuse memory from the pool.

Large buffers bypass the pool and allocate fresh memory.

* **Buffer Methods:**
* **toString()** → Converts buffer data into a string using the given encoding (default: 'utf-8').
* **write()** → Writes a string to the buffer at the specified offset using the given encoding.
* **toJSON ()** → Returns a JSON representation of the buffer (used automatically by JSON.stringify).
* **subarray()** → Returns a new view (not a copy) of the buffer between the given indexes.
* **copy()** → Copies data from one buffer to another.
* **includes()** → Checks if buffer contains the given value.
* **fill()** → Fills the buffer with the specified value.
* **readInt8(offset)** → Reads a signed 8-bit integer from the buffer at the given offset.
* **readInt16LE(offset)** → Reads a signed 16-bit integer (little-endian) from the buffer.
* **readInt16BE(offset)** → Reads a signed 16-bit integer (big-endian) from the buffer.
* **readUInt32LE(offset)** → Reads an unsigned 32-bit integer (little-endian).
* **at(index)** → Returns the byte at the given index, supports negative indexing.
* **equals(otherBuffer)** → Checks if two buffers have exactly the same content.
* **Buffer Properties (Top 12)**
* **buffer** → The underlying ArrayBuffer object backing the buffer.
* **byteLength** → Size of the underlying ArrayBuffer in bytes.
* **byteOffset** → The byte offset of the buffer from its underlying ArrayBuffer.
* **length** → The number of bytes in the buffer.
* **parent** → Reference to the parent memory block (internal, rarely used).
* **BYTES\_PER\_ELEMENT** → Always 1, since Buffer stores raw bytes.
* **Buffer Limitations:**
* Fixed size: Once you create a buffer, its size (length in bytes) cannot be changed.
* High memory consumption: It may lead to memory exhaustion or crashes.
* **Encoding limitations:** Buffers only support a limited set of encodings (utf8, ascii, base64, hex, etc.).
* **Base-64 Encoding?**

Base64 is a character encoding that takes binary data and converts it into a string.

* **In Browser: btoa() and atob()**

btoa () → **Binary to ASCII** → converts a string to Base64 encoded string.

atob () → **ASCII to Binary** → decodes Base64 string back to original.

* **Base-64 rules:**
* At least 3 bytes (24 bits) should be there to work with.
* If 3 bytes (24 bits) are not there then base64 encoding will multiple zero to fill remining bytes.
* Data should be in the multiples of 3 bytes.

# : Event Driven Architecture:

* **CPU Operations (Computation-bound tasks)**
* Arithmetic calculations (addition, multiplication, hashing, encryption).
* Running loops, sorting data, searching.
* Compiling code.
* Image/video rendering.
* **I/O Operations (Input/Output-bound tasks**): In NodeJS we perform I/O operations asynchronous way.
* Reading/writing files from disk.
* Sending/receiving data over the network.
* Waiting for user input (keyboard, mouse).
* Accessing a database.
* **NodeJS?**

NodeJS is a single-threaded, event-driven, non-blocking, asynchronous I/O runtime that allows developers to build scalable and high-performance applications, particularly suited for I/O bound tasks as a file handling and networking requests.

* **Single-threaded:** Node.js runs on a **single main thread** (unlike Java/other languages that often create multiple threads for handling requests).

**What it means**:

Only **one thread** executes JavaScript code at a time.

But Node.js can still handle **many requests concurrently** because it does not block on I/O.

* **Event-driven:** Node.js works on an event loop mechanism.

**What it means:**

Everything is triggered by events (like “file finished reading”, “HTTP request received”).

Node listens for events and executes callback functions when they occur.

* **Non-blocking I/O asynchronous operations:** Non-blocking = Node.js does not “pause” execution while waiting for I/O tasks. Asynchronous = Operations are started, and the result is handled later via callbacks, promises, or async/await.

**What it means:**

CPU keeps working on other things instead of getting stuck.

Ideal for I/O bound tasks (like reading files, making API calls, DB queries).

* **Event-Driven Architecture (EDA) in Node.js**

Event-Driven Architecture is a **design pattern** where the flow of the application is determined by **events**.  
In Node.js, everything (HTTP requests, file reads, DB queries) works on events. It follows **event-driven architecture** (events → listeners → handlers).

* **Main Components in Node.js**
* **Event Emitter:**

An object that **creates (emits) events**.

Example: Node.js provides the built-in Event Emitter class.

Example emitters: HTTP server (req, res), file system streams, custom emitters.

* **Event Listener**

A function that **listens (waits)** for a specific event.

It “subscribes” to an event.

* **Event Handler**

The actual **callback function** that executes when the event occurs.

* **Methods Summary**
* **on (event, listener)** → Attaches listener, runs multiple times.
* **once (event, listener)** → Attaches listener, runs only once.
* **emit (event, ...args)** → Emits an event & calls its listeners.
* **removeListener (event, listener)** → Removes a specific listener.
* **removeAllListeners (event)** → Removes all listeners for an event.
* **listeners (event)** → Returns array of listeners for event.
* **eventNames ()** → Returns array of all event names.

# : Stream in NodeJS?

* **What is Stream NodeJS?**

In **Node.js**, a **stream** is an object that allows **reading data or writing data continuously**.  
Instead of loading the whole data in memory (like reading a big file at once), streams let you **process data chunk by chunk**.

**Example:**

* Reading a 2GB file → Instead of loading the whole file, Node.js reads it in small chunks (buffers) using a stream.
* This makes apps **fast, memory efficient, and scalable**.
* **Types of Streams in NodeJS?**

There are **4 main types of streams**:

* Readable Stream.
* Writable Stream.
* Duplex Stream.
* Transform Stream.
* **FS Module (Stream related method)**
* fs.createReadStream(path, options) → Creates a readable stream to read file data in chunks.
* fs.createWriteStream(path, options) → Creates a writable stream to write data into a file in chunks.

1. **What is Readable Stream?**

Readable streams are used to **read data** from a source, such as reading files or receiving HTTP requests. High water mark value by default 64 kb.

* **States of readable Stream:**

There a mainly four states of readable Streams.

* **Initial State:** Stream is created but not consuming data yet.
* **Flowing State:** Stream is automatically emitting data via 'data' events.
* **Paused State:** Stream is not flowing; data can only be read manually using. read ().
* **Ended State:** Stream has finished sending all data and emitted the 'end' event.
* **All Three steps are continued with all states.**
* **readableFlowing** → Shows the current flow mode of the stream (null, true, or false).
* **null** → Stream created but not yet consuming.
* **true** → Stream is flowing and emitting 'data'.
* **false** → Stream is paused and not emitting automatically.
* **readableEnded** → Becomes true once the stream has finished reading all data.
* **isPaused()** → Returns true if the stream is paused, otherwise false.
* **Readable Stream Methods:**
* **.read()** → Reads data from internal buffer.
* .pipe(destination) → Sends data directly to a writable/transform stream.
* .unpipe([destination]) → Stops piping to a specific/all destinations.
* **.pause()** → Temporarily stops emitting data events.
* .**resume()** → Resumes emitting data events after pause.
* **.destroy([error])** → Forcefully closes the stream and optionally passes an error.
* **setEncoding(encoding name):** set the Stream encoding.
* **Readable Stream properties:**
* **bytesRead →** How much data has been read by the stream so far.
* **readableHighWaterMark →** The buffer’s capacity limit (maximum bytes stored before pausing reads).
* **readableLength:**

1. **what is writable Stream?**

Writable streams are used to write data to a destination, such as writing to a file or sending HTTP responses. High water mark value 16 kb.

* **Writable Stream Methods:**
* **.write(chunk, [encoding], [callback])** → Writes data chunk into stream.
* **.end([chunk], [encoding], [callback])** → Finishes writing and closes the stream.
* .cork() → Buffers writes until .uncork() is called.
* .uncork() → Flushes all buffered writes (after cork).
* .destroy([error]) → Forcefully closes writable stream.
* **Writable Stream properties:**
* **writableHighWaterMark →** The buffer’s capacity limit (maximum bytes stored before pausing reads).
* **Writable Stream states:**
* **writable:** This is the default state. When you create a writable stream, it is ready to accept data.

**Syntax:**

**writeStream.writable**

* **writableCorked:** When you call .cork(), the stream starts **buffering data internally** instead of writing it immediately.

It will only write once you call .uncork().

This is useful for **performance optimization** batching multiple small writes into one larger write.

Syntax:

writeStream.writableCorked

1. **Duplex Stream?**

Duplex streams are streams that can both read and write data. These are useful for situations like network communication, where you both send and receive data.

* **Duplex Stream (Readable + Writable) Methods:**
* A Duplex stream has both readable + writable methods. So it supports everything from Readable and Writable (above).
* Example methods: .write(), .end(), .pipe(), .read(), .destroy()

1. **Transform Stream?**

Transform streams are a special type of duplex stream where the output is a transformation of the input. They modify or process the data as it passes through the stream, such as compressing or encrypting data.

* **Transform Stream (Special Duplex) Methods:**
* A Transform stream modifies the data while reading/writing.
* Has all Duplex methods + one special:
* .transform(chunk, encoding, callback) → Method to process and transform input data before pushing output.

# :

# :

# :

# :

# :