Admissions & Support:
support@mockattempt.com
Phone / WhatsApp: +91 95400 07917
Course Code: NODE • **Updated:** 04 Nov 2026
https://mockattempt.com/courses/node-js-training

NODE • BACKEND & CLOUD RUNTIME ENGINEERING **BEGINNER TO PRODUCTION NODE.JS BACKEND ARCHITECT**

100% LIVE INDUSTRY EXPERT COHORT + INTERNSHIP LETTER

Node js Training

Master production Node.js from V8 + Libuv Event Loop internals, Buffers, Backpressure Streams, FS & Crypto to Child Processes, Express.js REST APIs, MySQL2 Pools, MongoDB Aggregations & CI/CD — with runnable Node.js programs, 386 Topic Mock Tests (17,860 MCQs), and Spoken Voice + MockAttempt AI Interviews at ₹2,999 (Incl. 18% GST).

Duration: 16–18 Weeks • Live + Hands-On Projects (216 Hrs) • **Modules:** 16 • **Live Topics:** 53 • **Subtopics:** 424+ • **Labs & Tasks:** 106 • **Mocks / AI Vivas:** 386 / 386 • **Projects:** 16 Module + 1 Live Client Prod

All-Inclusive Direct Fee (Incl. 18% GST • Zero Hidden Charges): Total Payable: ₹2,999 (Net Tuition ₹2,542 + Included 18% GST ₹457) • **Split Option:** 100% Credit to Job Track • **Effective ROI:** ~₹14/Guided Hr (~₹57/Live Topic)

Faculty: 100% Active Working IT Professionals • **Live Batch Tracks (7:00 AM – 8:00 PM IST Flexible Slots):** Weekday Live Online Batch (Mon – Fri • 1 Hr/Day • 7:00 AM – 8:00 PM) (Monday to Friday (1 Hour Live Session / Day) • 07:00 AM – 08:00 PM IST (Students Can Choose Any 1-Hour Slot Based on Trainer Availability) • 18–20 Weeks Fast-Track) | Weekend Live Online Batch (Sat – Sun • 2 Hrs/Day • 7:00 AM – 8:00 PM) (Saturday & Sunday (2 Hours Live Session / Day) • 07:00 AM – 08:00 PM IST (Students Can Choose Any 2-Hour Slot Based on Trainer Availability) • 20–22 Weeks Weekend Track)

✓ **Trainer LeetCode & HackerRank Mentorship (JavaScript & TypeScript (Node.js 22)): Live trainer support to crack LeetCode & HackerRank problems, optimize complexity, and handle edge cases.**

Core Technologies & Engineering Stack: Node.js 22 LTS • V8 & Libuv Event Loop • Streams & Buffers • Express.js REST APIs • node:crypto & TLS • MySQL2 & MongoDB • node:test & CI/CD

1. PROGRAM MASTERY OUTCOMES & ELIGIBILITY

End-to-End Engineering Outcomes:

1. Master V8 JIT compilation, Libuv non-blocking Event Loop phases, process.nextTick vs Promises, and race-condition prevention
2. Process multi-gigabyte payloads in constant RAM using Binary Buffers, Readable/Writable/Transform Streams, pipeline() backpressure, and Zlib
3. Engineer hardened Express.js REST APIs with modular routers, timing-safe HMAC/TLS security, Child Process scaling, MySQL2 pools, and MongoDB indexes
4. Clear 386 Timed Mock Exams (17,500 Node.js MCQs) and 350 Spoken Voice + MockAttempt AI Backend Technical Interviews
5. Non-blocking Event Loop architecture, Streams, Worker Threads, and clean Controller → Service → Repository layers.
6. Crack LeetCode and HackerRank problems in JavaScript & TypeScript (Node.js 22) with live trainer mentorship on algorithmic/query patterns, edge cases, and optimal complexity.
7. Schema-validated REST APIs with centralized error middleware, idempotency keys, and consistent HTTP status contracts.
8. Defense-in-depth security: HttpOnly JWT rotation, RBAC middleware, Helmet headers, CORS policies, and rate limiting.

Eligibility & Prerequisites: Working knowledge of core JavaScript (functions, objects, arrays, Promises, async/await) • Any laptop/PC running Node.js 20/22 LTS, npm, and VS Code • No prior Node js industry experience required — Module 1 starts from environment setup, core mental models, and plain-English fundamentals. • Ideal for B.Tech / BCA / MCA / B.Sc IT students, non-CS beginners, career switchers, and working developers upgrading to 2026 production standards. • Commitment to attend live Weekday or Weekend classes, complete daily hands-on lab checklists, and practice topic-wise Mock Tests & AI Interviews.

2. DETAILED ZERO-TO-EXPERT MODULE, TOPIC & SUBTOPIC SYLLABUS (16 MODULES • 53 LIVE TOPICS • 424+ SUBTOPICS)

Progressive Zero-to-Expert Mastery Stages: Modules 1–4: Stage 1: Day-1 Foundations & Core Mental Models (16 Topics / 128 Subtopics) • Modules 5–8: Stage 2: Applied Engineering, Data & API Workflows (14 Topics / 112 Subtopics) • Modules 9–12: Stage 3: Internals, Concurrency, Security & Performance (11 Topics / 88 Subtopics) • Modules 13–16: Stage 4: Production Architecture, Testing, CI/CD & Expert Mastery (12 Topics / 96 Subtopics)

MODULE 1 OF 16 • STAGE 1 • 14 LIVE + LAB HOURS • 4 TOPICS (32 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 1: Node.js Runtime Architecture, V8 Engine, Libuv, NPM & ESM vs CommonJS

Understand why single-threaded event-driven Node.js handles tens of thousands of concurrent connections, V8 JIT compilation & heap spaces, npm SemVer lockfiles, and ES Modules (`import`) vs CommonJS (`require`).

Objectives: ✓ Node.js Architecture — Event-Driven Non-Blocking I/O, Single JS Thread & Libuv Overview — What Node.js is (C++ runtime embedding V8 + Libuv), why I/O-bound APIs scale effortlessly without thread-per-request overhead, and when CPU-heavy tasks must be offloaded. • ✓ Google V8 JavaScript Engine — Ignition Bytecode, TurboFan JIT & Generational Heap GC — How V8 parses JS into AST, Ignition interpreter bytecode, TurboFan optimizing JIT compiler & deoptimization bailouts, New Space (Scavenge) vs Old Space (Mark-Sweep-Compact) GC, and `node:v8`. • ✓ NPM & Package Governance — `package.json`, SemVer (`^` vs `~`), `package-lock.json` & `npm ci` — Authoring `package.json` (`engines`, `exports`, `scripts`), Semantic Versioning (`MAJOR.MINOR.PATCH`), caret `^1.2.3` vs tilde `~1.2.3`, deterministic `npm ci` in CI/CD, and `npm audit`. • ✓ CommonJS (`require`/`module.exports`) vs ES Modules (`import`/`export`) in Node.js — Synchronous dynamic `require()` vs asynchronous static ESM `import`, `"type": "module"` vs `.mjs`/`.cjs`, `import.meta.dirname` replacing `__dirname`, top-level `await`, and module caching. • ✓ Build, verify, and ship Module Capstone: Production ESM Node.js Workspace & V8 Heap Telemetry CLI.

1.1 Node.js Architecture — Event-Driven Non-Blocking I/O, Single JS Thread & Libuv Overview

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Non-blocking event-driven I/O & Single JS main thread • Core Syntax, CLI & Production APIs: node:http, node:fs/promises, Libuv event demultiplexer & 4-thread UV_THREADPOOL_SIZE • Deep-Dive Mechanics: Libuv thread pool & Epoll/kqueue/IOCP • Advanced Patterns: I/O-bound vs CPU-bound & Node: built-in prefix • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Non-Blocking Runtime Inspection & Async Micro-Benchmark (`server.mjs`) • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Node.js Architecture

Purpose: To understand and apply Node.js Architecture — Event-Driven Non-Blocking I/O, Single JS Thread & Libuv Overview in real Node js systems — specifically: What Node.js is (C...

What Topic Is Next → Next Topic 1.2: Google V8 JavaScript Engine — Ignition Bytecode, TurboFan JIT & Generational Heap GC

1.2 Google V8 JavaScript Engine — Ignition Bytecode, TurboFan JIT & Generational Heap GC

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Ignition bytecode & TurboFan JIT & New Space vs Old Space GC • Core Syntax, CLI & Production APIs: import v8 from "node:v8"; v8.getHeapStatistics(), --max-old-space-size=4096 • Deep-Dive Mechanics: Hidden classes (Shapes) & V8.getHeapStatistics() • Advanced Patterns: --max-old-space-size & Memory leak heap snapshots • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Inspecting V8 Heap Limit & Live Memory Allocation (`v8-stats.mjs`) • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Google V8 JavaScript Engine

Purpose: To understand and apply Google V8 JavaScript Engine — Ignition Bytecode, TurboFan JIT & Generational Heap GC in real Node js systems — specifically: How V8 parses JS into...

What Topic Is Next → Next Topic 1.3: NPM & Package Governance — `package.json`, SemVer (`^` vs `~`), `package-lock.json` & `npm ci`

1.3 NPM & Package Governance — `package.json`, SemVer (`^` vs `~`), `package-lock.json` & `npm ci`

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: SemVer ^caret vs ~tilde & Package-lock.json determinism • Core Syntax, CLI & Production APIs: npm ci, npm audit, package.json ("type": "module", "exports", "engines") • Deep-Dive Mechanics: Npm ci --omit=dev & Npm audit vulnerabilities • Advanced Patterns: Dependencies vs devDependencies & Package.json exports map • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Semantic Version Caret Range Compatibility Checker • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Package Governance

Purpose: To understand and apply NPM & Package Governance — `package.json`, SemVer (`^` vs `~`), `package-lock.json` & `npm ci` in real Node js systems — specifically: Authoring `...

What Topic Is Next → Next Topic 1.4: CommonJS (`require`/`module.exports`) vs ES Modules (`import`/`export`) in Node.js

1.4 CommonJS (`require`/`module.exports`) vs ES Modules (`import`/`export`) in Node.js

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: ESM static import vs CJS require & Top-level await in ESM • Core Syntax, CLI & Production APIs: import/export, import.meta.url, import.meta.dirname, createRequire(import.meta.url) • Deep-Dive Mechanics: Import.meta.dirname & .mjs and .cjs extensions • Advanced Patterns: Module singleton cache & Dynamic import() • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Modern ES Module Named & Default Exports with `import.meta.url` • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for CommonJS (`require`/`module.exports`) vs ES Modules...

Purpose: To understand and apply CommonJS (`require`/`module.exports`) vs ES Modules (`import`/`export`) in Node.js in real Node js systems — specifically: Synchronous dynamic `re...

What Topic Is Next → Module 1 Capstone (Production ESM Node.js Workspace & V8 Heap Telemetry CLI) → Next Topic 2.1: Node.js Global Scope — `globalThis`, `URL`, `fetch`, `AbortController`, `structuredClone` & Module Scope

Module 1 Hands-On Capstone Project (100 Marks): Production ESM Node.js Workspace & V8 Heap Telemetry CLI — Build an ES Module (`"type": "module"`) Node.js CLI diagnostics tool that prints Node/V8 versions, live heap space breakdown via `node:v8`, dependency lockfile audit checks, and non-blocking...

MODULE 2 OF 16 • STAGE 1 • 12 LIVE + LAB HOURS • 4 TOPICS (32 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 2: Node.js Globals, `process` Lifecycle, Signals, `node:os` Telemetry & Timers

Master `globalThis`, `process.env` validation, graceful `SIGTERM`/`SIGINT` shutdown hooks, `node:os` CPU/memory inspection, and Event Loop timers (`setImmediate`, `setTimeout`, `AbortSignal.timeout`).

Objectives: ✓ Node.js Global Scope — `globalThis`, `URL`, `fetch`, `AbortController`, `structuredClone` & Module Scope — Top-level `const`/`let` are file-scoped (never attached to `globalThis`), built-in web-compatible globals (`fetch`, `URL`, `TextEncoder`, `crypto`, `structuredClone`), and why mutating `global` breaks test isolation. • ✓ The `node:process` Object — `process.env`, `argv`, `memoryUsage()`, Exit Codes & `SIGTERM` Shutdown — Reading environment config (`node --env-file=.env`), CLI arguments (`util.parseArgs`), RSS vs heapUsed in `process.memoryUsage()`, `process.nextTick`, and graceful container shutdown on `SIGTERM`. • ✓ Operating System Module (`node:os`) — `availableParallelism()`, Memory, Endianness & Host Metrics — Querying container-aware CPU limits via `os.availableParallelism()`, `os.totalmem()`/`os.freemem()`, `os.uptime()`, `os.tmpdir()`, `os.EOL`, and platform-portable newline/temp handling. • ✓ Node.js Timers (`node:timers`, `timers/promises`) — `setImmediate`, `setTimeout`, `unref()` & Cancellation — Timers phase (`setTimeout`/`setInterval`) vs Check phase (`setImmediate`), `timers/promises` async `setTimeout(ms, val, { signal })`, and `.unref()` so background cache-sweepers don't hang shutdown. • ✓ Build, verify, and ship Module Capstone: Zero-Downtime Graceful Shutdown Supervisor & Host Telemetry Agent.

2.1 Node.js Global Scope — `globalThis`, `URL`, `fetch`, `AbortController`, `structuredClone` & Module Scope

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: GlobalThis vs module scope & Cross-request state leak trap • Core Syntax, CLI & Production APIs: globalThis, structuredClone(obj), AbortController, AbortSignal.timeout(ms), TextEncoder • Deep-Dive Mechanics: StructuredClone deep copy & Native fetch & AbortSignal • Advanced Patterns: URL & URLSearchParams & No window/document • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Deep Cloning Nested Config Safely with `structuredClone` • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Node.js Global Scope

Purpose: To understand and apply Node.js Global Scope — `globalThis`, `URL`, `fetch`, `AbortController`, `structuredClone` & Module Scope in real Node js systems — specifically: T...

What Topic Is Next → Next Topic 2.2: The `node:process` Object — `process.env`, `argv`, `memoryUsage()`, Exit Codes & `SIGTERM` Shutdown

2.2 The `node:process` Object — `process.env`, `argv`, `memoryUsage()`, Exit Codes & `SIGTERM` Shutdown

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Process.env string coercion trap & Graceful SIGTERM shutdown • Core Syntax, CLI & Production APIs: process.env, process.memoryUsage(), process.on("SIGTERM", fn), node --env-file=.env • Deep-Dive Mechanics: Process.memoryUsage() RSS vs heapUsed & Node --env-file=.env • Advanced Patterns: Process.nextTick queue & POSIX exit codes • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Safe Environment Parsing & Process Memory Snapshot • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for The `node:process` Object

Purpose: To understand and apply The `node:process` Object — `process.env`, `argv`, `memoryUsage()`, Exit Codes & `SIGTERM` Shutdown in real Node js systems — specifically: Readin...

What Topic Is Next → Next Topic 2.3: Operating System Module (`node:os`) — `availableParallelism()`, Memory, Endianness & Host Metrics

2.3 Operating System Module (`node:os`) — `availableParallelism()`, Memory, Endianness & Host Metrics

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Os.availableParallelism() vs os.cpus() & Docker cgroup CPU limits • Core Syntax, CLI & Production APIs: import os from "node:os"; os.availableParallelism(), os.freemem(), os.tmpdir(), os.EOL • Deep-Dive Mechanics: Os.freemem() & os.totalmem() & Os.tmpdir() scratch files • Advanced Patterns: Os.EOL cross-platform newlines & Os.loadavg() • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Container-Aware Worker Sizing & Host Memory Check (`os-probe.mjs`) • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Operating System Module (`node:os`)

Purpose: To understand and apply Operating System Module (`node:os`) — `availableParallelism()`, Memory, Endianness & Host Metrics in real Node js systems — specifically: Querying...

What Topic Is Next → Next Topic 2.4: Node.js Timers (`node:timers`, `timers/promises`) — `setImmediate`, `setTimeout`, `unref()` & Cancellation

2.4 Node.js Timers (`node:timers`, `timers/promises`) — `setImmediate`, `setTimeout`, `unref()` & Cancellation

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: SetImmediate vs setTimeout in I/O & Node:timers/promises async delay • Core Syntax, CLI & Production APIs: import { setTimeout } from "node:timers/promises"; setImmediate(), timer.unref() • Deep-Dive Mechanics: Timer.unref() clean exit & AbortController timer cancellation • Advanced Patterns: Timers phase vs Check phase & ClearInterval leak prevention • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Cancelable Promise Timer & `.unref()` Background Heartbeat • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Node.js Timers (`node:timers`, `timers/promises`)

Purpose: To understand and apply Node.js Timers (`node:timers`, `timers/promises`) — `setImmediate`, `setTimeout`, `unref()` & Cancellation in real Node js systems — specifically:...

What Topic Is Next → Module 2 Capstone (Zero-Downtime Graceful Shutdown Supervisor & Host Telemetry Agent) → Next Topic 3.1: Error-First Callbacks `(err, result)`, Callback Hell, Zalgo Anti-Pattern & `util.promisify`

Module 2 Hands-On Capstone Project (100 Marks): Zero-Downtime Graceful Shutdown Supervisor & Host Telemetry Agent — Build a background Node.js health daemon that validates required `process.env` keys at boot, reports `node:os` load averages & free memory every interval using `.unref()` timers, and drains...

MODULE 3 OF 16 • STAGE 1 • 14 LIVE + LAB HOURS • 4 TOPICS (32 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 3: Event Loop Phases, Callbacks vs Promises, `EventEmitter` Pub/Sub & Concurrency Safety

Trace every Libuv Event Loop phase, convert error-first callbacks with `util.promisify`, build memory-safe `EventEmitter` buses, centralize async error recovery, and eliminate async race conditions.

Objectives: ✓ Error-First Callbacks `(err, result)`, Callback Hell, Zalgo Anti-Pattern & `util.promisify` — Node's classic `(err, value)` convention, why invoking a callback synchronously on cache hit and asynchronously on cache miss ("releasing Zalgo") causes subtle bugs, and bridging to Promises via `util.promisify`. • ✓ `node:events` & `EventEmitter` — Synchronous Dispatch, `once()`, `error` Events & Listener Leaks — Pub/Sub inside Node (`on`, `once`, `emit`, `off`), `events.once(emitter, "ready", { signal })` Promise helper, why `emitter.emit()` invokes listeners synchronously, and `MaxListenersExceededWarning`. • ✓ Error Architecture in Node.js — Operational vs Programmer Errors, `Error.cause` & Unhandled Rejections — System errors (`ENOENT`, `ECONNREFUSED`, `EADDRINUSE`), custom `AppError` classes with HTTP status & `{ cause: err }` chaining, operational vs programmer crashes, and `unhandledRejection` telemetry. • ✓ Race Conditions in Single-Threaded Node.js — Interleaved `await` Points, TOCTOU & Atomic Locks — Why "single-threaded JS" STILL suffers race conditions across `await` boundaries (read-modify-write balance bugs, file check-then-open TOCTOU), atomic SQL updates (`SET balance = balance - ? WHERE balance >= ?`), and Promise mutex queues. • ✓ Build, verify, and ship Module Capstone: Resilient EventEmitter Order Bus with Keyed Async Mutex & Dead-Letter Error Queue.

3.1 Error-First Callbacks `(err, result)`, Callback Hell, Zalgo Anti-Pattern & `util.promisify` [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Error-first (err, data) contract & Missing return callback(err) bug • Core Syntax, CLI & Production APIs: import { promisify } from "node:util"; fs.promises, queueMicrotask() • Deep-Dive Mechanics: Releasing Zalgo sync/async mix & Util.promisify() • Advanced Patterns: Callback pyramid vs async/await & Promise.all parallel I/O • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Converting Legacy Error-First Callback Function with `util.promisify` • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Error-First Callbacks `(err, result)`

Purpose: To understand and apply Error-First Callbacks `(err, result)`, Callback Hell, Zalgo Anti-Pattern & `util.promisify` in real Node js systems — specifically: Node's classic...

What Topic Is Next → Next Topic 3.2: `node:events` & `EventEmitter` — Synchronous Dispatch, `once()`, `error` Events & Listener Leaks

3.2 `node:events` & `EventEmitter` — Synchronous Dispatch, `once()`, `error` Events & Listener Leaks [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Unhandled "error" event crash & Synchronous emit() execution • Core Syntax, CLI & Production APIs: import { EventEmitter, once } from "node:events"; emitter.on(), emitter.once(), captureRejec... • Deep-Dive Mechanics: MaxListenersExceededWarning leak & CaptureRejections: true • Advanced Patterns: Await once(emitter, event) & AbortSignal listener removal • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Safe `EventEmitter` with `captureRejections` & `once()` • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for `node:events`

Purpose: To understand and apply `node:events` & `EventEmitter` — Synchronous Dispatch, `once()`, `error` Events & Listener Leaks in real Node js systems — specifically: Pub/Sub i...

What Topic Is Next → Next Topic 3.3: Error Architecture in Node.js — Operational vs Programmer Errors, `Error.cause` & Unhandled Rejections

3.3 Error Architecture in Node.js — Operational vs Programmer Errors, `Error.cause` & Unhandled Rejections [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Operational vs programmer errors & Error { cause } chaining • Core Syntax, CLI & Production APIs: new Error("msg", { cause }), err.code ("ENOENT"), process.on("unhandledRejection") • Deep-Dive Mechanics: POSIX err.code (ENOENT/EADDRINUSE) & UnhandledRejection termination • Advanced Patterns: Try/catch with await & Graceful restart after fatal state • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Structured Domain Error with Native ES2022 `{ cause }` Chaining • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Error Architecture in Node.js

Purpose: To understand and apply Error Architecture in Node.js — Operational vs Programmer Errors, `Error.cause` & Unhandled Rejections in real Node js systems — specifically: Sys...

What Topic Is Next → Next Topic 3.4: Race Conditions in Single-Threaded Node.js — Interleaved `await` Points, TOCTOU & Atomic Locks

3.4 Race Conditions in Single-Threaded Node.js — Interleaved `await` Points, TOCTOU & Atomic Locks [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Await interleaving race condition & Read-modify-write lost update • Core Syntax, CLI & Production APIs: Keyed Promise Queue / Mutex, Atomic DB UPDATE ... WHERE, fs.open(path, "wx") exclusive flag • Deep-Dive Mechanics: TOCTOU file race (use wx flag) & Keyed Promise chain mutex • Advanced Patterns: Atomic SQL conditional update & Idempotency keys • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Preventing Double-Spend Across Concurrent `await` Calls with a Keyed Queue • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Race Conditions in Single-Threaded Node.js

Purpose: To understand and apply Race Conditions in Single-Threaded Node.js — Interleaved `await` Points, TOCTOU & Atomic Locks in real Node js systems — specifically: Why "single...

What Topic Is Next → Module 3 Capstone (Resilient EventEmitter Order Bus with Keyed Async Mutex & Dead-Letter Error Queue) → Next Topic 4.1: Node.js `Buffer` (`node:buffer`) — `Buffer.alloc` vs `allocUnsafe`, UTF-8/Base64/Hex & `Uint8Array`

Module 3 Hands-On Capstone Project (100 Marks): Resilient EventEmitter Order Bus with Keyed Async Mutex & Dead-Letter Error Queue — Engineer an in-memory Node.js Order Processing Bus extending `EventEmitter` that deduplicates concurrent wallet debits per `userId` using a keyed async mutex, handles `'error'` events safely...

MODULE 4 OF 16 • STAGE 1 • 14 LIVE + LAB HOURS • 4 TOPICS (32 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 4: Binary `Buffer`, Backpressure `Stream` Pipelines, Async `node:fs` & Safe `node:path`

Manipulate raw binary bytes safely with `Buffer`, process 10 GB files in 32 MB RAM using `stream/promises` `pipeline()`, master `node:fs/promises`, and block path-traversal attacks with `node:path`.

Objectives: ✓ Node.js `Buffer` (`node:buffer`) — `Buffer.alloc` vs `allocUnsafe`, UTF-8/Base64/Hex & `Uint8Array` — Fixed-length byte sequences backed by `ArrayBuffer`, why `new Buffer()` is deprecated, `Buffer.alloc(n)` zero-filling vs `Buffer.allocUnsafe(n)` uninitialized pool memory, `Buffer.concat`, and

Base64URL encoding. • ✓ Node.js Streams (`node:stream`) — Readable, Writable, Transform, Backpressure & `pipeline()` — Four stream types (Readable, Writable, Duplex, Transform), `highWaterMark` buffering, why `.pipe()` leaks file descriptors on error while `stream/promises` `pipeline()` cleans up everything, and `for await (const chunk of readable)`. • ✓ File System (`node:fs/promises`) — Non-Blocking Read/Write, `createReadStream`, Atomic Rename & `stat` — Using `fs/promises` (`readFile`, `writeFile`, `mkdir({ recursive: true })`, `stat`, `opendir` async iterator) vs `fs.createReadStream` for large files, and why `*Sync` methods are forbidden inside request handlers. • ✓ `node:path` Security & Cross-Platform Paths — `join` vs `resolve`, `parse` & Blocking Path Traversal — POSIX `/` vs Windows `\` separators, `path.join()` concatenation vs `path.resolve()` right-to-left absolute resolution, `path.basename`/`extname`, and defending static/upload endpoints against `../../` traversal. • ✓ Build, verify, and ship Module Capstone: Constant-Memory Streaming CSV/NDJSON Sanitizer & Path-Safe File Vault.

4.1 Node.js `Buffer` (`node:buffer`) — `Buffer.alloc` vs `allocUnsafe`, UTF-8/Base64/Hex & `Uint8Array` [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Buffer.alloc zero-filled safety & Buffer.allocUnsafe memory leak risk • Core Syntax, CLI & Production APIs: Buffer.alloc(size), Buffer.from(data, enc), Buffer.concat(list), buf.toString("base64url") • Deep-Dive Mechanics: UTF-8 vs Base64URL vs Hex & Uint8Array compatibility • Advanced Patterns: Subarray shared memory view & Buffer.concat() • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Binary UTF-8 & Base64URL Token Encoding with `node:buffer` • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Node.js `Buffer` (`node:buffer`)

Purpose: To understand and apply Node.js `Buffer` (`node:buffer`) — `Buffer.alloc` vs `allocUnsafe`, UTF-8/Base64/Hex & `Uint8Array` in real Node js systems — specifically: Fixed-...

What Topic Is Next → Next Topic 4.2: Node.js Streams (`node:stream`) — Readable, Writable, Transform, Backpressure & `pipeline()`

4.2 Node.js Streams (`node:stream`) — Readable, Writable, Transform, Backpressure & `pipeline()` [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Pipeline() vs legacy .pipe() leak & Readable / Transform / Writable • Core Syntax, CLI & Production APIs: import { pipeline } from "node:stream/promises"; Readable, Transform, Writable • Deep-Dive Mechanics: HighWaterMark backpressure & Drain event flow control • Advanced Patterns: For await...of async iteration & ObjectMode streams • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: In-Memory `Readable` -> `Transform` -> `Writable` Pipeline (`stream-demo.mjs`) • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Node.js Streams (`node:stream`)

Purpose: To understand and apply Node.js Streams (`node:stream`) — Readable, Writable, Transform, Backpressure & `pipeline()` in real Node js systems — specifically: Four stream t...

What Topic Is Next → Next Topic 4.3: File System (`node:fs/promises`) — Non-Blocking Read/Write, `createReadStream`, Atomic Rename & `stat`

4.3 File System (`node:fs/promises`) — Non-Blocking Read/Write, `createReadStream`, Atomic Rename & `stat` [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Node:fs/promises async API & Never use Sync in HTTP routes • Core Syntax, CLI & Production APIs: import fs from "node:fs/promises"; createReadStream, mkdir({ recursive: true }), rename() • Deep-Dive Mechanics: CreateReadStream for large files & Atomic temp-write + fs.rename • Advanced Patterns: Opendir async iterator & Handling err.code ENOENT • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Async Temp Directory & Atomic File Write Verification • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for File System (`node:fs/promises`)

Purpose: To understand and apply File System (`node:fs/promises`) — Non-Blocking Read/Write, `createReadStream`, Atomic Rename & `stat` in real Node js systems — specifically: Usi...

What Topic Is Next → Next Topic 4.4: `node:path` Security & Cross-Platform Paths — `join` vs `resolve`, `parse` & Blocking Path Traversal

4.4 `node:path` Security & Cross-Platform Paths — `join` vs `resolve`, `parse` & Blocking Path Traversal [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Path.join vs path.resolve(/reset) & Blocking ../ directory traversal • Core Syntax, CLI & Production APIs: import path from "node:path"; path.join(), path.resolve(), path.relative(), path.basename() • Deep-Dive Mechanics: StartsWith(baseDir + path.sep) & POSIX / vs Windows \\ • Advanced Patterns: Path.parse & path.extname & Import.meta.dirname • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Path-Traversal Guard Function Using `path.resolve` & `path.sep` • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for `node:path` Security

Purpose: To understand and apply `node:path` Security & Cross-Platform Paths — `join` vs `resolve`, `parse` & Blocking Path Traversal in real Node js systems — specifically: POSIX...

What Topic Is Next → Module 4 Capstone (Constant-Memory Streaming CSV/NDJSON Sanitizer & Path-Safe File Vault) → Next Topic 5.1: Core Web Server Module (`node:http`) — `createServer`, Keep-Alive Agents & Socket Timeouts

Module 4 Hands-On Capstone Project (100 Marks): Constant-Memory Streaming CSV/NDJSON Sanitizer & Path-Safe File Vault — Create a streaming Node.js ETL processor using `node:stream/promises` `pipeline()` and a custom `Transform` stream that redacts PII emails line-by-line from arbitrarily large log files in fl...

MODULE 5 OF 16 • STAGE 2 • 14 LIVE + LAB HOURS • 4 TOPICS (32 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 5: Native HTTP Web Servers (`node:http`), Request/Response Streams, URL Query & `node:zlib`

Build zero-framework HTTP/1.1 servers with `node:http`, parse streaming JSON request bodies with size limits, query URLs via WHATWG `URL`/`URLSearchParams`, and compress responses using Gzip/Brotli (`node:zlib`).

Objectives: ✓ Core Web Server Module (`node:http`) — `createServer`, Keep-Alive Agents & Socket Timeouts — How `http.createServer((req, res) => ...)` multiplexes thousands of persistent Keep-Alive TCP connections on one thread, configuring `requestTimeout` / `headersTimeout`, and connection pooling with `http.Agent`. • ✓ Handling HTTP Requests & Responses — Streaming Body Chunks, Status Codes & Header Safety — Reading chunked request bodies via `for await (const chunk of req)`, enforcing max byte caps (`413`), setting headers before body flush (`ERR_HTTP_HEADERS_SENT`), and streaming JSON/SSE responses. • ✓ URL & Query String Parsing — Modern WHATWG `URL` & `URLSearchParams` vs Legacy `node:querystring` — Why WHATWG `new URL(req.url,

"http://localhost")` and `URLSearchParams` supersede legacy `url.parse()` / `node:querystring`, handling repeated keys (`getAll("tag")`), safe RFC-3986 encoding, and pagination validation. • ✓ HTTP & Stream Compression (`node:zlib`) — Gzip, Deflate, Brotli & Zip-Bomb Protection — Compressing JSON/HTML payloads by 70%–85% using `zlib.createGzip()` and `zlib.createBrotliCompress()`, promisified `gzip`/`gunzip`, `maxOutputLength` zip-bomb defense, and avoiding synchronous `gzipSync` on the Event Loop. • ✓ Build, verify, and ship Module Capstone: Zero-Dependency `node:http` JSON Micro-Router with Brotli Compression & Body Size Guard.

5.1 Core Web Server Module (`node:http`) — `createServer`, Keep-Alive Agents & Socket Timeouts [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Http.createServer lifecycle & IncomingMessage & ServerResponse streams • Core Syntax, CLI & Production APIs: import http from "node:http"; http.createServer(), server.headersTimeout, server.close() • Deep-Dive Mechanics: HTTP Keep-Alive reuse & HeadersTimeout & requestTimeout • Advanced Patterns: Server.close() graceful drain & Ephemeral port listen(0) • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Ephemeral-Port `node:http` JSON Health Server Round-Trip Test • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Core Web Server Module (`node:http`)

Purpose: To understand and apply Core Web Server Module (`node:http`) — `createServer`, Keep-Alive Agents & Socket Timeouts in real Node js systems — specifically: How `http.creat...

What Topic Is Next → Next Topic 5.2: Handling HTTP Requests & Responses — Streaming Body Chunks, Status Codes & Header Safety

5.2 Handling HTTP Requests & Responses — Streaming Body Chunks, Status Codes & Header Safety [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: ERR_HTTP_HEADERS_SENT guard & Multi-byte UTF-8 chunk boundary • Core Syntax, CLI & Production APIs: for await (const chunk of req), res.writeHead(status, headers), res.headersSent, res.end() • Deep-Dive Mechanics: 413 Payload Too Large cap & Buffer.concat(chunks) • Advanced Patterns: Res.headersSent check & Content-Type & status codes • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Bounded UTF-8 Safe Request Stream Body Reader Helper • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Handling HTTP Requests

Purpose: To understand and apply Handling HTTP Requests & Responses — Streaming Body Chunks, Status Codes & Header Safety in real Node js systems — specifically: Reading chunked r...

What Topic Is Next → Next Topic 5.3: URL & Query String Parsing — Modern WHATWG `URL` & `URLSearchParams` vs Legacy `node:querystring`

5.3 URL & Query String Parsing — Modern WHATWG `URL` & `URLSearchParams` vs Legacy `node:querystring` [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: WHATWG new URL(req.url, base) & URLSearchParams get vs getAll • Core Syntax, CLI & Production APIs: const u = new URL(req.url, "http://localhost"); u.searchParams.get("page"), u.searchParams.g... • Deep-Dive Mechanics: Deprecated url.parse SSRF risk & Automatic percent-encoding • Advanced Patterns: Bounded page/limit clamping & Iterable searchParams • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Safe Query Filtering & Pagination Clamping with `URLSearchParams` • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Query String Parsing

Purpose: To understand and apply URL & Query String Parsing — Modern WHATWG `URL` & `URLSearchParams` vs Legacy `node:querystring` in real Node js systems — specifically: Why WHAT...

What Topic Is Next → Next Topic 5.4: HTTP & Stream Compression (`node:zlib`) — Gzip, Deflate, Brotli & Zip-Bomb Protection

5.4 HTTP & Stream Compression (`node:zlib`) — Gzip, Deflate, Brotli & Zip-Bomb Protection [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Brotli vs Gzip compression & Pipeline(src, createGzip(), res) • Core Syntax, CLI & Production APIs: import zlib from "node:zlib"; createBrotliCompress(), createGzip(), gunzip(buf, { maxOutputL... • Deep-Dive Mechanics: MaxOutputLength zip-bomb guard & Never use gzipSync in routes • Advanced Patterns: Vary: Accept-Encoding header & Skip already-compressed media • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Async Gzip Compression & Bounded Decompression with `node:zlib` • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for HTTP

Purpose: To understand and apply HTTP & Stream Compression (`node:zlib`) — Gzip, Deflate, Brotli & Zip-Bomb Protection in real Node js systems — specifically: Compressing JSON/HTM...

What Topic Is Next → Module 5 Capstone (Zero-Dependency `node:http` JSON Micro-Router with Brotli Compression & Body Size Guard) → Next Topic 6.1: `node:crypto` Essentials — `randomUUID`, Salted `scrypt` Passwords, `createHmac` & `AES-256-GCM`

Module 5 Hands-On Capstone Project (100 Marks): Zero-Dependency `node:http` JSON Micro-Router with Brotli Compression & Body Size Guard — Build a lightweight native `node:http` JSON server supporting `/health`, `/courses?category=...&page=...` via `new URL(req.url, base)`, bounded JSON body parsing (rejecting > 1 MB with `413...

MODULE 6 OF 16 • STAGE 2 • 12 LIVE + LAB HOURS • 3 TOPICS (24 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 6: Applied Cryptography (`node:crypto`), Constant-Time HMAC Verification & TLS/SSL Security

Hash passwords safely with salted `crypto.scrypt`, encrypt secrets with authenticated `AES-256-GCM`, defeat timing side-channel attacks using `crypto.timingSafeEqual()`, and configure TLS 1.3 / mTLS servers.

Objectives: ✓ `node:crypto` Essentials — `randomUUID`, Salted `scrypt` Passwords, `createHmac` & `AES-256-GCM` — CSPRNG token generation (`crypto.randomBytes`, `crypto.randomUUID`), why plain `SHA-256` is broken for passwords (use memory-hard `scrypt`), authenticated encryption (`aes-256-gcm` + `getAuthTag()`), and WebCrypto `crypto.subtle`. • ✓ Constant-Time Secret Comparison (`crypto.timingSafeEqual`) — Defeating Timing Side-Channel Attacks — Why standard `sigA === sigB` short-circuits on the first mismatched byte (leaking expected HMAC characters nanosecond-by-nanosecond), how `crypto.timingSafeEqual(bufA, bufB)` runs in constant time, and guarding against length `RangeError`. • ✓ TLS/SSL & HTTPS in Node.js (`node:tls`, `node:https`) — TLS 1.3, Certificates, ALPN & mTLS — TLS handshake fundamentals, X.509 certificate chains, configuring `minVersion: "TLSv1.2"`, mutual TLS (`requestCert: true`, `rejectUnauthorized: true`) for service-to-service zero-trust, and why disabling cert checks is fatal. • ✓ Build, verify, and ship Module Capstone: Constant-Time Webhook Signature Verifier, `scrypt` Auth Vault & AES-256-GCM Token Engine.

6.1 `node:crypto` Essentials — `randomUUID`, Salted `scrypt` Passwords, `createHmac` & `AES-256-GCM` [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Crypto.randomBytes & randomUUID & Memory-hard scrypt + random salt • Core Syntax, CLI & Production APIs: import crypto from "node:crypto"; randomUUID(), scrypt(), createHmac("sha256", key), createC... • Deep-Dive Mechanics: Never use Math.random for tokens & AES-256-GCM IV + AuthTag • Advanced Patterns: HMAC-SHA256 message integrity & Libuv async key derivation • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Authenticated `AES-256-GCM` Encrypt & Tamper-Checked Decrypt (`crypto-gcm.mjs`) • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for `node:crypto` Essentials

Purpose: To understand and apply `node:crypto` Essentials — `randomUUID`, Salted `scrypt` Passwords, `createHmac` & `AES-256-GCM` in real Node js systems — specifically: CSPRNG to...

What Topic Is Next → Next Topic 6.2: Constant-Time Secret Comparison (`crypto.timingSafeEqual`) — Defeating Timing Side-Channel Attacks

6.2 Constant-Time Secret Comparison (`crypto.timingSafeEqual`) — Defeating Timing Side-Channel Attacks [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: === short-circuit timing leak & Crypto.timingSafeEqual(a, b) • Core Syntax, CLI & Production APIs: crypto.timingSafeEqual(bufA, bufB) with prior bufA.byteLength === bufB.byteLength check • Deep-Dive Mechanics: RangeError on unequal byteLength & HMAC webhook verification • Advanced Patterns: Buffer.from(hex, "hex") & Constant-time XOR comparison • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Production Webhook HMAC-SHA256 Verifier Using `timingSafeEqual` • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Constant-Time Secret Comparison (`crypto.timingSafeE...

Purpose: To understand and apply Constant-Time Secret Comparison (`crypto.timingSafeEqual`) — Defeating Timing Side-Channel Attacks in real Node js systems — specifically: Why sta...

What Topic Is Next → Next Topic 6.3: TLS/SSL & HTTPS in Node.js (`node:tls`, `node:https`) — TLS 1.3, Certificates, ALPN & mTLS

6.3 TLS/SSL & HTTPS in Node.js (`node:tls`, `node:https`) — TLS 1.3, Certificates, ALPN & mTLS [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: TLS 1.3 1-RTT forward secrecy & Never set NODE_TLS_REJECT_UNAUTHORIZED=0 • Core Syntax, CLI & Production APIs: import tls from "node:tls"; import https from "node:https"; minVersion: "TLSv1.2", rejectUna... • Deep-Dive Mechanics: NODE_EXTRA_CA_CERTS safe CA trust & Mutual TLS (mTLS) client certs • Advanced Patterns: SNI & ALPN negotiation & HSTS Strict-Transport-Security • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Inspecting Supported TLS Cipher Suites & Default Minimum TLS Version • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for TLS/SSL

Purpose: To understand and apply TLS/SSL & HTTPS in Node.js (`node:tls`, `node:https`) — TLS 1.3, Certificates, ALPN & mTLS in real Node js systems — specifically: TLS handshake f...

What Topic Is Next → Module 6 Capstone (Constant-Time Webhook Signature Verifier, `scrypt` Auth Vault & AES-256-GCM Token Engine) → Next Topic 7.1: `node:child_process` Security & Mechanics — `execFile`, `spawn`, Exit Signals & Avoiding Shell Injection

Module 6 Hands-On Capstone Project (100 Marks): Constant-Time Webhook Signature Verifier, `scrypt` Auth Vault & AES-256-GCM Token Engine — Implement a zero-dependency Node.js security library using `node:crypto` providing salted async `scrypt` password hashing, authenticated `aes-256-gcm` encryption (with 12-byte IV + 16-byte A...

MODULE 7 OF 16 • STAGE 2 • 12 LIVE + LAB HOURS • 3 TOPICS (24 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 7: Multi-Core Scaling — `node:child_process`, `spawn` vs `fork` IPC & Custom NPM Modules

Scale beyond a single CPU core safely with `node:child_process` (`spawn`, `execFile`, `fork` IPC), avoid shell-injection vulnerabilities, and package clean dual/ESM custom Node.js modules.

Objectives: ✓ `node:child_process` Security & Mechanics — `execFile`, `spawn`, Exit Signals & Avoiding Shell Injection — The four process APIs (`spawn`, `fork`, `execFile`, `exec`), streaming `child.stdout`/`stderr`, `maxBuffer` limits on `exec`/`execFile`, and why shell interpolation causes Remote Code Execution (RCE). • ✓ `spawn()` vs `fork()` vs `worker_threads` — Stdio Streams, V8 Process Isolation & IPC Channels — Comparing `spawn()` (any system binary via stdio streams), `fork()` (dedicated Node.js child process with built-in IPC `process.send()` / `child.on("message")` and isolated V8 heap), and `node:worker_threads` (shared memory `SharedArrayBuffer` for CPU math). • ✓ Creating Custom Node.js Modules — `exports` Subpath Encapsulation, Barrel Pitfalls & Dependency Injection — Designing cohesive domain modules, `package.json` `"exports"` map hiding private `./internal/*` files, factory dependency injection for easy unit testing, and SemVer publishing hygiene. • ✓ Build, verify, and ship Module Capstone: Multi-Process PDF/Report IPC Worker Pool (`fork`) & Reusable `@maa/logger` Custom Package.

7.1 `node:child_process` Security & Mechanics — `execFile`, `spawn`, Exit Signals & Avoiding Shell Injection [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: RCE command injection in exec() & ExecFile(bin, [args]) array safety • Core Syntax, CLI & Production APIs: import { execFile, spawn } from "node:child_process"; promisify(execFile) • Deep-Dive Mechanics: Shell: false default in spawn & ERR_CHILD_PROCESS_STDIO_MAXBUFFER • Advanced Patterns: Exit code vs signal termination & AbortSignal process kill • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Safe Argument-Array Subprocess Execution with `execFile` • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for `node:child_process` Security

Purpose: To understand and apply `node:child_process` Security & Mechanics — `execFile`, `spawn`, Exit Signals & Avoiding Shell Injection in real Node js systems — specifically: T...

What Topic Is Next → Next Topic 7.2: `spawn()` vs `fork()` vs `worker_threads` — Stdio Streams, V8 Process Isolation & IPC Channels

7.2 `spawn()` vs `fork()` vs `worker_threads` — Stdio Streams, V8 Process Isolation & IPC Channels [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Spawn stdio streams any binary & Fork isolated V8 + IPC channel • Core Syntax, CLI & Production APIs: spawn(cmd, args), fork(modulePath) + child.send(msg), new Worker(url) (node:worker_threads) • Deep-Dive Mechanics: Worker_threads CPU math pool & Child.send & message event • Advanced Patterns:

Memory cost of fork vs thread & Fixed worker pool sizing • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Workload Classifier Choosing `spawn` vs `fork` vs `worker_threads` • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for `spawn()` vs `fork()` vs `worker_threads`

Purpose: To understand and apply `spawn()` vs `fork()` vs `worker_threads` — Stdio Streams, V8 Process Isolation & IPC Channels in real Node js systems — specifically: Comparing `...

What Topic Is Next → Next Topic 7.3: Creating Custom Node.js Modules — `exports` Subpath Encapsulation, Barrel Pitfalls & Dependency Injection

7.3 Creating Custom Node.js Modules — `exports` Subpath Encapsulation, Barrel Pitfalls & Dependency Injection [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Package.json subpath exports & ERR_PACKAGE_PATH_NOT_EXPORTED • Core Syntax, CLI & Production APIs: export function createService(deps) {}, package.json "exports": { ".": "./src/index.js" } • Deep-Dive Mechanics: Factory dependency injection & Avoiding circular import TDZ • Advanced Patterns: Stateless pure exports & Npm pack dry-run inspection • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Testable Custom Rate-Limiter Module Using Factory Dependency Injection • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Creating Custom Node.js Modules

Purpose: To understand and apply Creating Custom Node.js Modules — `exports` Subpath Encapsulation, Barrel Pitfalls & Dependency Injection in real Node js systems — specifically:...

What Topic Is Next → Module 7 Capstone (Multi-Process PDF/Report IPC Worker Pool (`fork`) & Reusable `@maa/logger` Custom Package) → Next Topic 8.1: Express.js Modular Routing — `express.Router()`, Route Params, Versioning & RESTful Status Codes

Module 7 Hands-On Capstone Project (100 Marks): Multi-Process PDF/Report IPC Worker Pool (`fork`) & Reusable `@maa/logger` Custom Package — Build a bounded `child_process.fork()` worker pool with parent<->child IPC messaging, crash auto-respawn, and per-job timeouts so heavy report generation never blocks HTTP latency, packaged...

MODULE 8 OF 16 • STAGE 2 • 16 LIVE + LAB HOURS • 4 TOPICS (32 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 8: Express.js Production REST API Engineering — Routing, Middleware, Request Validation & Uploads

Architect layered Express.js APIs with modular `express.Router()`, authentication/rate-limit/error middleware chains, schema-validated `req.params`/`req.query`/`req.body`, and hardened `multer` file uploads.

Objectives: ✓ Express.js Modular Routing — `express.Router()`, Route Params, Versioning & RESTful Status Codes — Mounting domain routers (`app.use("/api/v1/courses", coursesRouter)`), route specificity order (static `/courses/featured` BEFORE dynamic `/courses/:id`!), `mergeParams: true` for nested resources, and idempotent HTTP verbs (`GET`, `POST`, `PATCH`, `PUT`, `DELETE`). • ✓ Express.js Middleware Pipeline — `(req, res, next)`, Execution Order, CORS, Helmet & 4-Arg Error Handler — Onion/pipeline flow of `app.use()`, hanging requests when neither `next()` nor `res.json()` is called, security headers (`helmet`), `express.json({ limit: "100kb" })`, and registering the 4-argument `(err, req, res, next)` handler last. • ✓ Inspecting & Validating `req` in Express — `params`, `query`, `body`, `get()`, `app.set("trust proxy")` & Zod — Distinguishing `req.params` (URL path segments — always strings), `req.query` (query parameters), `req.body` (parsed payload), case-insensitive `req.get("authorization")`, `trust proxy` behind Nginx/Cloudflare, and runtime schema validation. • ✓ Secure File Uploads in Express (`multer` & Streaming Multipart) — Size Caps, UUID Names & Magic Bytes — Handling `multipart/form-data` with `multer`, configuring `limits: { fileSize, files }`, generating `crypto.randomUUID()` filenames outside the public webroot, checking file magic bytes (`%PDF-`, PNG `89 50 4E 47`), and streaming large uploads directly to S3/R2. • ✓ Build, verify, and ship Module Capstone: Hardened Express.js Course & Assignment Portal REST API (`/api/v1`).

8.1 Express.js Modular Routing — `express.Router()`, Route Params, Versioning & RESTful Status Codes [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Top-to-bottom route specificity & Express.Router({ mergeParams: true }) • Core Syntax, CLI & Production APIs: const router = express.Router({ mergeParams: true }); router.get("/:slug", handler) • Deep-Dive Mechanics: API versioning /api/v1 & 201 Created vs 204 No Content • Advanced Patterns: 404 catch-all handler placement & Express 5 async error forwarding • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Deterministic Top-to-Bottom Route Matcher Modeling `express.Router()` • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Express.js Modular Routing

Purpose: To understand and apply Express.js Modular Routing — `express.Router()`, Route Params, Versioning & RESTful Status Codes in real Node js systems — specifically: Mounting...

What Topic Is Next → Next Topic 8.2: Express.js Middleware Pipeline — `(req, res, next)`, Execution Order, CORS, Helmet & 4-Arg Error Handler

8.2 Express.js Middleware Pipeline — `(req, res, next)`, Execution Order, CORS, Helmet & 4-Arg Error Handler [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: (req, res, next) pipeline order & Hanging request on missing next() • Core Syntax, CLI & Production APIs: app.use(express.json({ limit: "100kb" })), next(), next(err), (err, req, res, next) => {} • Deep-Dive Mechanics: 4-arg (err, req, res, next) arity & Express.json({ limit }) DoS guard • Advanced Patterns: Helmet security headers & CORS & Request-scoped res.locals • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Composable Express-Style Middleware Runner Verified End-to-End • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Express.js Middleware Pipeline

Purpose: To understand and apply Express.js Middleware Pipeline — `(req, res, next)`, Execution Order, CORS, Helmet & 4-Arg Error Handler in real Node js systems — specifically: O...

What Topic Is Next → Next Topic 8.3: Inspecting & Validating `req` in Express — `params`, `query`, `body`, `get()`, `app.set("trust proxy")` & Zod

8.3 Inspecting & Validating `req` in Express — `params`, `query`, `body`, `get()`, `app.set("trust proxy")` & Zod [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Mass Assignment vulnerability guard & Zod/schema whitelist validation • Core Syntax, CLI & Production APIs: req.params, req.query, req.body, req.get("Authorization"), app.set("trust proxy", 1) • Deep-Dive Mechanics: Req.params vs req.query vs req.body & Case-insensitive req.get(header) • Advanced Patterns: Trust proxy for accurate req.ip & 422 validation error payload • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Strict Whitelist Payload Validator Blocking Mass-Assignment Fields • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... •

Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Inspecting

Purpose: To understand and apply Inspecting & Validating `req` in Express — `params`, `query`, `body`, `get()`, `app.set("trust proxy")` & Zod in real Node js systems — specifical...

What Topic Is Next → Next Topic 8.4: Secure File Uploads in Express (`multer` & Streaming Multipart) — Size Caps, UUID Names & Magic Bytes

8.4 Secure File Uploads in Express (`multer` & Streaming Multipart) — Size Caps, UUID Names & Magic Bytes [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Limits: { fileSize, files } cap & Never trust file.originalname • Core Syntax, CLI & Production APIs: multer({ storage, limits: { fileSize: 5 * 1024 * 1024, files: 1 }, fileFilter }) • Deep-Dive Mechanics: Magic-byte signature check & UUID storage filename • Advanced Patterns: Disk/stream vs memoryStorage OOM & MulterError 413 handling • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: PDF & PNG Magic-Byte Signature Inspector (`magic-check.mjs`) • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Secure File Uploads in Express (`multer` & Streaming...

Purpose: To understand and apply Secure File Uploads in Express (`multer` & Streaming Multipart) — Size Caps, UUID Names & Magic Bytes in real Node js systems — specifically: Hand...

What Topic Is Next → Module 8 Capstone (Hardened Express.js Course & Assignment Portal REST API (`/api/v1`)) → Next Topic 9.1: MySQL/MariaDB in Node.js (`mysql2/promise`) — `createPool` vs `createConnection` & Transaction Safety

Module 8 Hands-On Capstone Project (100 Marks): Hardened Express.js Course & Assignment Portal REST API (`/api/v1`) — Build a production-structured Express.js REST API featuring modular `/api/v1/courses` & `/api/v1/submissions` routers, request-ID logging, Bearer JWT auth middleware, strict payload validati...

MODULE 9 OF 16 • STAGE 3 • 12 LIVE + LAB HOURS • 3 TOPICS (24 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 9: Relational & Document Persistence — `mysql2/promise` Connection Pools & MongoDB Queries

Connect Node.js backends to MySQL/MariaDB with pooled prepared statements (`mysql2/promise`) and query MongoDB collections efficiently using compound indexes, projections, and cursor pagination.

Objectives: ✓ MySQL/MariaDB in Node.js (`mysql2/promise`) — `createPool` vs `createConnection` & Transaction Safety — Why production servers use `mysql.createPool({ connectionLimit, waitForConnections, queueLimit })` instead of single `createConnection`, transactional `const conn = await pool.getConnection()`, and guaranteed `conn.release()` in `finally`. • ✓ Querying MySQL Records Safely — `?` Prepared Placeholders, Keyset Pagination & Streaming Large Result Sets — Preventing SQL Injection with `pool.execute("SELECT id, title FROM courses WHERE slug = ?", [slug])`, avoiding `SELECT *`, keyset cursor pagination (`WHERE id > ? ORDER BY id LIMIT 20`), and streaming large exports. • ✓ MongoDB Queries in Node.js — Compound Indexes (ESR Rule), Projections, Aggregations & Blocking NoSQL Injection — Querying with the official `mongodb` Node driver / Mongoose (`find`, `projection`, `.lean()`), Equality-Sort-Range (ESR) compound index design, `.explain("executionStats")` `IXSCAN` vs `COLLSCAN`, and blocking `{"$ne": null}` NoSQL operator injection. • ✓ Build, verify, and ship Module Capstone: Polyglot Analytics & Enrollment Data Layer (`mysql2` Transactions + MongoDB Indexed Queries).

9.1 MySQL/MariaDB in Node.js (`mysql2/promise`) — `createPool` vs `createConnection` & Transaction Safety [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: CreatePool vs createConnection & Finally { conn.release() } rule • Core Syntax, CLI & Production APIs: import mysql from "mysql2/promise"; mysql.createPool({ connectionLimit: 10 }), conn.beginTra... • Deep-Dive Mechanics: ConnectionLimit & queueLimit & BeginTransaction / commit / rollback • Advanced Patterns: Pool.execute prepared cache & Graceful pool.end() on SIGTERM • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Transactional Pool Borrow & Guaranteed `finally` Release Simulation • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for MySQL/MariaDB in Node.js (`mysql2/promise`)

Purpose: To understand and apply MySQL/MariaDB in Node.js (`mysql2/promise`) — `createPool` vs `createConnection` & Transaction Safety in real Node js systems — specifically: Why...

What Topic Is Next → Next Topic 9.2: Querying MySQL Records Safely — `?` Prepared Placeholders, Keyset Pagination & Streaming Large Result Sets

9.2 Querying MySQL Records Safely — `?` Prepared Placeholders, Keyset Pagination & Streaming Large Result Sets [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: ? parameterized prepared statements & SQL Injection prevention • Core Syntax, CLI & Production APIs: const [rows] = await pool.execute("SELECT id, title FROM courses WHERE price <= ? LIMIT ?",... • Deep-Dive Mechanics: Allowlist dynamic ORDER BY columns & Keyset pagination vs slow OFFSET • Advanced Patterns: Explicit column projection & Streaming million-row queries • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Parameterized SQL Builder & Strict Sort-Column Allowlist Helper • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Querying MySQL Records Safely

Purpose: To understand and apply Querying MySQL Records Safely — `?` Prepared Placeholders, Keyset Pagination & Streaming Large Result Sets in real Node js systems — specifically:...

What Topic Is Next → Next Topic 9.3: MongoDB Queries in Node.js — Compound Indexes (ESR Rule), Projections, Aggregations & Blocking NoSQL Injection

9.3 MongoDB Queries in Node.js — Compound Indexes (ESR Rule), Projections, Aggregations & Blocking NoSQL Injection [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: NoSQL $ne/$gt injection defense & ESR compound index rule • Core Syntax, CLI & Production APIs: collection.find({ category: String(cat) }, { projection: { title: 1, fee: 1 } }).sort({ fee:... • Deep-Dive Mechanics: IXSCAN vs COLLSCAN explain() & Field projection bandwidth save • Advanced Patterns: Mongoose .lean() plain objects & Aggregation $match early stage • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: NoSQL Operator-Injection Sanitizer & ESR Filter Builder • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for MongoDB Queries in Node.js

Purpose: To understand and apply MongoDB Queries in Node.js — Compound Indexes (ESR Rule), Projections, Aggregations & Blocking NoSQL Injection in real Node js systems — specifica...

What Topic Is Next → Module 9 Capstone (Polyglot Analytics & Enrollment Data Layer (`mysql2` Transactions + MongoDB Indexed Queries)) → Next Topic 10.1: Automated Unit & Integration Testing (`node:test` & `node:assert/strict`) — Mocks, Async Rejections & Coverage

Module 9 Hands-On Capstone Project (100 Marks): Polyglot Analytics & Enrollment Data Layer (`mysql2` Transactions + MongoDB Indexed Queries) — Implement a transactional MySQL/MariaDB Enrollment Repository (`BEGIN` -> seat lock -> student group insert -> `COMMIT`/`ROLLBACK` with guaranteed `conn.release()` in `finally`) paired with...

MODULE 10 OF 16 • STAGE 3 • 12 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 10: Native Automated Testing (`node:test`) & Production CI/CD Deployment Pipelines

Write fast zero-dependency unit & integration tests using Node's built-in `node:test` + `node:assert/strict` runner, mock clocks/timers cleanly, and ship zero-downtime Docker & GitHub Actions CI/CD pipelines.

Objectives: ✓ Automated Unit & Integration Testing (`node:test` & `node:assert/strict`) — Mocks, Async Rejections & Coverage — Node's built-in test runner (`node --test`), `describe`/`it`/`beforeEach`, `assert.strictEqual` / `assert.deepStrictEqual` / `await assert.rejects()`, `t.mock.fn()` & `t.mock.timers`, and isolating external HTTP/DB boundaries. • ✓ CI/CD & Production Node.js Operations — GitHub Actions, `npm ci --omit=dev`, Non-Root Docker & Health Probes — Designing automated CI gates (lint -> typecheck -> `npm ci` -> `node --test` -> security audit), multi-stage `node:22-alpine` Docker images running as non-root `USER node`, `NODE_ENV=production`, `/healthz` vs `/readyz` probes, and zero-downtime rolling deploys. • ✓ Build, verify, and ship Module Capstone: Zero-Flake `node:test` Suite, Code Coverage Gate & Multi-Stage Production CI/CD Workflow.

10.1 Automated Unit & Integration Testing (`node:test` & `node:assert/strict`) — Mocks, Async Rejections & Coverage [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Built-in node:test runner & Node:assert/strict equality • Core Syntax, CLI & Production APIs: import test from "node:test"; import assert from "node:assert/strict"; await assert.rejects(... • Deep-Dive Mechanics: Await assert.rejects() async check & T.mock.fn() & mock.timers • Advanced Patterns: Arrange-Act-Assert isolation & Ephemeral port 0 API testing • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Strict Assertion & Async Rejection Verification Using `node:assert/strict` • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Automated Unit

Purpose: To understand and apply Automated Unit & Integration Testing (`node:test` & `node:assert/strict`) — Mocks, Async Rejections & Coverage in real Node js systems — specifica...

What Topic Is Next → Next Topic 10.2: CI/CD & Production Node.js Operations — GitHub Actions, `npm ci --omit=dev`, Non-Root Docker & Health Probes

10.2 CI/CD & Production Node.js Operations — GitHub Actions, `npm ci --omit=dev`, Non-Root Docker & Health Probes [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: CMD ["node", ...] direct PID 1 SIGTERM & Npm ci --omit=dev slim image • Core Syntax, CLI & Production APIs: npm ci --omit=dev, NODE_ENV=production, USER node, CMD ["node", "src/server.mjs"] • Deep-Dive Mechanics: Non-root USER node container & NODE_ENV=production optimizations • Advanced Patterns: /healthz liveness vs /readyz readiness & GitHub Actions Node matrix CI • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Readiness Probe Evaluator Checking DB Pool & Shutdown Drain Flag • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for CI/CD

Purpose: To understand and apply CI/CD & Production Node.js Operations — GitHub Actions, `npm ci --omit=dev`, Non-Root Docker & Health Probes in real Node js systems — specificall...

What Topic Is Next → Module 10 Capstone (Zero-Flake `node:test` Suite, Code Coverage Gate & Multi-Stage Production CI/CD Workflow) → Next Topic 11.1: Short-Lived JWT Access + Rotating HttpOnly Refresh Tokens

Module 10 Hands-On Capstone Project (100 Marks): Zero-Flake `node:test` Suite, Code Coverage Gate & Multi-Stage Production CI/CD Workflow — Create a comprehensive `node --test --experimental-test-coverage` suite covering async services, mocked timers, and ephemeral-port HTTP endpoints using `node:assert/strict`, paired with a ha...

MODULE 11 OF 16 • STAGE 3 • 14 LIVE + LAB HOURS • 3 TOPICS (24 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Production Auth, RBAC Permissions, Rate Limiting & OWASP Hardening

Production-grade Node js Training engineering module covering Production Auth, RBAC Permissions, Rate Limiting & OWASP Hardening with hands-on labs and capstone build.

Objectives: ✓ Short-Lived JWT Access + Rotating HttpOnly Refresh Tokens — Prevent token theft and revoke compromised user sessions instantly across devices. • ✓ Role & Scope RBAC Middleware plus Sliding-Window Rate Limiting — Enforce tenant isolation and block credential-stuffing or scraping floods at the edge. • ✓ Helmet Headers, Strict Schema Validation, CORS & HMAC Signatures — Eliminate NoSQL/SQL injection, prototype pollution, and forged payment webhooks. • ✓ Build, verify, and ship Module Capstone: Multi-Tenant Auth & HMAC Webhook Verification Gateway.

11.1 Short-Lived JWT Access + Rotating HttpOnly Refresh Tokens [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Edge • Integration with Role & Scope RBAC Middleware plus Sliding • Live Debugging, Telemetry & Technical Interview Trade • What Is Short • In Node js Training, Short • Even if you come from a non • Runnable Code Implementation (JAVASCRIPT)

Purpose: To understand and apply Short-Lived JWT Access + Rotating HttpOnly Refresh Tokens in real Node js systems — specifically: Prevent token theft and revoke compromised user...

What Topic Is Next → Next Topic 11.2: Role & Scope RBAC Middleware plus Sliding-Window Rate Limiting

11.2 Role & Scope RBAC Middleware plus Sliding-Window Rate Limiting [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Scope RBAC Middleware plus Sliding • Integration with Helmet Headers, Strict Schema Validation, CORS & HMAC Signatures • Live Debugging, Telemetry & Technical Interview Trade • What Is Role & Scope RBAC Middleware plus Sliding • In Node js Training, Role & Scope RBAC Middleware plus Sliding • Even if you come from a non • Runnable Code Implementation (JAVASCRIPT)

Purpose: To understand and apply Role & Scope RBAC Middleware plus Sliding-Window Rate Limiting in real Node js systems — specifically: Enforce tenant isolation and block credenti...

What Topic Is Next → Next Topic 11.3: Helmet Headers, Strict Schema Validation, CORS & HMAC Signatures

11.3 Helmet Headers, Strict Schema Validation, CORS & HMAC Signatures [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Strict Schema Validation • Integration with Short • Live Debugging, Telemetry & Technical Interview Trade • What Is Helmet Headers, Strict Schema Validation, CORS & HMAC Signatures & Why Industry Teams Use It • In Node js Training, Helmet Headers, Strict Schema Validation, CORS & HMAC Signatures exists for one clear real • Even if you come from a non • Runnable Code Implementation (JAVASCRIPT)

Purpose: To understand and apply Helmet Headers, Strict Schema Validation, CORS & HMAC Signatures in real Node js systems — specifically: Eliminate NoSQL/SQL injection, prototype...

What Topic Is Next → Module 11 Capstone (Multi-Tenant Auth & HMAC Webhook Verification Gateway) → Next Topic 12.1: Cache-Aside Pattern, TTL Jitter & Targeted Key Invalidation

Module 11 Hands-On Capstone Project (100 Marks): Multi-Tenant Auth & HMAC Webhook Verification Gateway — Design, implement, test, and document 'Multi-Tenant Auth & HMAC Webhook Verification Gateway' applying every production concept from Module 11 (Production Auth, RBAC Permissions, Rate Limiti...

MODULE 12 OF 16 • STAGE 3 • 14 LIVE + LAB HOURS • 3 TOPICS (24 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Redis Caching, Distributed Locks & BullMQ Background Worker Queues

Production-grade Node js Training engineering module covering Redis Caching, Distributed Locks & BullMQ Background Worker Queues with hands-on labs and capstone build.

Objectives: ✓ Cache-Aside Pattern, TTL Jitter & Targeted Key Invalidation — Drop p95 database read latency from 120ms to 3ms on high-traffic endpoints. • ✓ BullMQ Job Producers, Concurrency Workers & Dead-Letter Queues — Offload slow PDF/email/webhook work from HTTP threads with exponential retry safety. • ✓ Atomic Redis Locks & Idempotency-Key Payment Protection — Guarantee retried POST requests never double-charge or double-book inventory. • ✓ Build, verify, and ship Module Capstone: Async Invoice, Email & Webhook Retry Engine with BullMQ.

12.1 Cache-Aside Pattern, TTL Jitter & Targeted Key Invalidation [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of TTL Jitter • Integration with BullMQ Job Producers, Concurrency Workers & Dead • Live Debugging, Telemetry & Technical Interview Trade • What Is Cache • In Node js Training, Cache • Even if you come from a non • Runnable Code Implementation (JAVASCRIPT)

Purpose: To understand and apply Cache-Aside Pattern, TTL Jitter & Targeted Key Invalidation in real Node js systems — specifically: Drop p95 database read latency from 120ms to 3...

What Topic Is Next → Next Topic 12.2: BullMQ Job Producers, Concurrency Workers & Dead-Letter Queues

12.2 BullMQ Job Producers, Concurrency Workers & Dead-Letter Queues [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Concurrency Workers • Integration with Atomic Redis Locks & Idempotency • Live Debugging, Telemetry & Technical Interview Trade • What Is BullMQ Job Producers, Concurrency Workers & Dead • In Node js Training, BullMQ Job Producers, Concurrency Workers & Dead • Even if you come from a non • Runnable Code Implementation (JAVASCRIPT)

Purpose: To understand and apply BullMQ Job Producers, Concurrency Workers & Dead-Letter Queues in real Node js systems — specifically: Offload slow PDF/email/webhook work from HT...

What Topic Is Next → Next Topic 12.3: Atomic Redis Locks & Idempotency-Key Payment Protection

12.3 Atomic Redis Locks & Idempotency-Key Payment Protection [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Idempotency • Integration with Cache • Live Debugging, Telemetry & Technical Interview Trade • What Is Atomic Redis Locks & Idempotency • In Node js Training, Atomic Redis Locks & Idempotency • Even if you come from a non • Runnable Code Implementation (JAVASCRIPT)

Purpose: To understand and apply Atomic Redis Locks & Idempotency-Key Payment Protection in real Node js systems — specifically: Guarantee retried POST requests never double-charg...

What Topic Is Next → Module 12 Capstone (Async Invoice, Email & Webhook Retry Engine with BullMQ) → Next Topic 13.1: stream.pipeline() Backpressure for Multi-GB Uploads & Exports

Module 12 Hands-On Capstone Project (100 Marks): Async Invoice, Email & Webhook Retry Engine with BullMQ — Design, implement, test, and document 'Async Invoice, Email & Webhook Retry Engine with BullMQ' applying every production concept from Module 12 (Redis Caching, Distributed Locks & BullMQ Ba...

MODULE 13 OF 16 • STAGE 4 • 14 LIVE + LAB HOURS • 3 TOPICS (24 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Streams, Worker Threads, Cluster Mode & Graceful Shutdown

Production-grade Node js Training engineering module covering Streams, Worker Threads, Cluster Mode & Graceful Shutdown with hands-on labs and capstone build.

Objectives: ✓ stream.pipeline() Backpressure for Multi-GB Uploads & Exports — Stream millions of records to S3 or HTTP responses without V8 heap exhaustion. • ✓ Worker Threads for CPU-Bound Crypto/Parsing & Cluster Scaling — Utilize all CPU cores while keeping the main Node.js event loop < 10ms lag. • ✓ SIGTERM

Graceful Drain, /healthz & /readyz Readiness Probes — Deploy new Node releases with zero dropped requests or half-written transactions. • ✓ Build, verify, and ship Module Capstone: Constant-Memory Streaming CSV Importer & Multi-Core API.

13.1 stream.pipeline() Backpressure for Multi-GB Uploads & Exports [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Exports • Integration with Worker Threads for CPU • Live Debugging, Telemetry & Technical Interview Trade • What Is stream.pipeline() Backpressure for Multi • In Node js Training, stream.pipeline() Backpressure for Multi • Even if you come from a non • Runnable Code Implementation (JAVASCRIPT)

Purpose: To understand and apply stream.pipeline() Backpressure for Multi-GB Uploads & Exports in real Node js systems — specifically: Stream millions of records to S3 or HTTP res...

What Topic Is Next → Next Topic 13.2: Worker Threads for CPU-Bound Crypto/Parsing & Cluster Scaling

13.2 Worker Threads for CPU-Bound Crypto/Parsing & Cluster Scaling [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Cluster Scaling • Integration with SIGTERM Graceful Drain, /healthz & /readyz Readiness Probes • Live Debugging, Telemetry & Technical Interview Trade • What Is Worker Threads for CPU • In Node js Training, Worker Threads for CPU • Even if you come from a non • Runnable Code Implementation (JAVASCRIPT)

Purpose: To understand and apply Worker Threads for CPU-Bound Crypto/Parsing & Cluster Scaling in real Node js systems — specifically: Utilize all CPU cores while keeping the main...

What Topic Is Next → Next Topic 13.3: SIGTERM Graceful Drain, /healthz & /readyz Readiness Probes

13.3 SIGTERM Graceful Drain, /healthz & /readyz Readiness Probes [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of /healthz • Integration with stream.pipeline() Backpressure for Multi • Live Debugging, Telemetry & Technical Interview Trade • What Is SIGTERM Graceful Drain, /healthz & /readyz Readiness Probes & Why Industry Teams Use It • In Node js Training, SIGTERM Graceful Drain, /healthz & /readyz Readiness Probes exists for one clear real • Even if you come from a non • Runnable Code Implementation (JAVASCRIPT)

Purpose: To understand and apply SIGTERM Graceful Drain, /healthz & /readyz Readiness Probes in real Node js systems — specifically: Deploy new Node releases with zero dropped req...

What Topic Is Next → Module 13 Capstone (Constant-Memory Streaming CSV Importer & Multi-Core API) → Next Topic 14.1: Integration Testing Express Routers & Auth Flows with Supertest

Module 13 Hands-On Capstone Project (100 Marks): Constant-Memory Streaming CSV Importer & Multi-Core API — Design, implement, test, and document 'Constant-Memory Streaming CSV Importer & Multi-Core API' applying every production concept from Module 13 (Streams, Worker Threads, Cluster Mode & Grac...

MODULE 14 OF 16 • STAGE 4 • 14 LIVE + LAB HOURS • 3 TOPICS (24 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Automated API Testing (Supertest), Structured Pino Logs & Docker CI

Production-grade Node js Training engineering module covering Automated API Testing (Supertest), Structured Pino Logs & Docker CI with hands-on labs and capstone build.

Objectives: ✓ Integration Testing Express Routers & Auth Flows with Supertest — Verify status codes, DB side effects, and validation errors automatically on every PR. • ✓ AsyncLocalStorage Correlation IDs & Structured Pino JSON Logs — Trace every log entry across async calls and redact sensitive secrets automatically. • ✓ Multi-Stage Node Alpine Dockerfiles & GitHub Actions Deployment — Ship tiny, non-root Node.js production containers behind TLS reverse proxies. • ✓ Build, verify, and ship Module Capstone: Containerized Node.js Microservice with Supertest CI Suite.

14.1 Integration Testing Express Routers & Auth Flows with Supertest [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Auth Flows with Supertest • Integration with AsyncLocalStorage Correlation IDs & Structured Pino JSON Logs • Live Debugging, Telemetry & Technical Interview Trade • What Is Integration Testing Express Routers & Auth Flows with Supertest & Why Industry Teams Use It • In Node js Training, Integration Testing Express Routers & Auth Flows with Supertest exists for one clear real • Even if you come from a non • Runnable Code Implementation (JAVASCRIPT)

Purpose: To understand and apply Integration Testing Express Routers & Auth Flows with Supertest in real Node js systems — specifically: Verify status codes, DB side effects, and...

What Topic Is Next → Next Topic 14.2: AsyncLocalStorage Correlation IDs & Structured Pino JSON Logs

14.2 AsyncLocalStorage Correlation IDs & Structured Pino JSON Logs [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Structured Pino JSON Logs • Integration with Multi • Live Debugging, Telemetry & Technical Interview Trade • What Is AsyncLocalStorage Correlation IDs & Structured Pino JSON Logs & Why Industry Teams Use It • In Node js Training, AsyncLocalStorage Correlation IDs & Structured Pino JSON Logs exists for one clear real • Even if you come from a non • Runnable Code Implementation (JAVASCRIPT)

Purpose: To understand and apply AsyncLocalStorage Correlation IDs & Structured Pino JSON Logs in real Node js systems — specifically: Trace every log entry across async calls and...

What Topic Is Next → Next Topic 14.3: Multi-Stage Node Alpine Dockerfiles & GitHub Actions Deployment

14.3 Multi-Stage Node Alpine Dockerfiles & GitHub Actions Deployment [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of GitHub Actions Deployment • Integration with Integration Testing Express Routers & Auth Flows with Supertest • Live Debugging, Telemetry & Technical Interview Trade • What Is Multi • In Node js Training, Multi • Even if you come from a non • Runnable Code Implementation (JAVASCRIPT)

Purpose: To understand and apply Multi-Stage Node Alpine Dockerfiles & GitHub Actions Deployment in real Node js systems — specifically: Ship tiny, non-root Node.js production con...

What Topic Is Next → Module 14 Capstone (Containerized Node.js Microservice with Supertest CI Suite) → Next Topic 15.1: Strict Request Schema Validation, Correlation IDs & Structured Pino JSON Logs

Module 14 Hands-On Capstone Project (100 Marks): Containerized Node.js Microservice with Supertest CI Suite — Design, implement, test, and document 'Containerized Node.js Microservice with Supertest CI Suite' applying every production concept from Module 14 (Automated API Testing (Supertest), Struct...

MODULE 15 OF 16 • STAGE 4 • 14 LIVE + LAB HOURS • 3 TOPICS (24 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Resilient Node.js Micro-APIs: Idempotency, Queues & Outbound Timeouts

Self-explanatory list-and-bullet production mastery module covering Resilient Node.js Micro-APIs: Idempotency, Queues & Outbound Timeouts in Node js Training, paired with hands-on labs, capstone repository, Topic Mock Tests, and Spoken MockAttempt AI Viva.

Objectives: ✓ Strict Request Schema Validation, Correlation IDs & Structured Pino JSON Logs — Purpose: Reject malformed payloads at the edge and trace every request across logs in milliseconds. • ✓ Outbound HTTP Timeouts, Circuit Breakers & Exponential Backoff Retries — Purpose: Prevent one slow third-party payment or SMS gateway from hanging your entire Node.js event loop. • ✓ Background Queue Workers, Dead-Letter Retries & Graceful SIGTERM Drain — Purpose: Finish in-flight jobs safely during container restarts without losing customer tasks. • ✓ Build, verify, and document 'Fault-Tolerant Webhook & Background Job Worker Service in Node.js' with clean Git commits and edge-case protection. • ✓ Clear 100% of Topic Timed Mock Tests and Spoken Voice + MockAttempt AI Technical Interviews.

15.1 Strict Request Schema Validation, Correlation IDs & Structured Pino JSON Logs

[65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Strict Request Schema Validation, Correlation IDs & Structured Pino JSON L... • Performance Optimization, Complexity & Latency Impact of Strict Request Schema Validation, Correlation IDs & Struct... • Live Debugging, Stack • Runnable Code Implementation (JAVASCRIPT) • Hands • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Strict Request Schema Validation • Production Hardening & Security: HttpOnly rotating JWTs, Zod boundary schemas, Helmet & OWASP BOLA guards

Purpose: To understand and apply Strict Request Schema Validation, Correlation IDs & Structured Pino JSON Logs in real Node js systems — specifically: Reject malformed payloads at...

What Topic Is Next → Next Topic 15.2: Outbound HTTP Timeouts, Circuit Breakers & Exponential Backoff Retries

15.2 Outbound HTTP Timeouts, Circuit Breakers & Exponential Backoff Retries

[65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Outbound HTTP Timeouts, Circuit Breakers & Exponential Backoff Retries • Performance Optimization, Complexity & Latency Impact of Outbound HTTP Timeouts, Circuit Breakers & Exponential Bac... • Live Debugging, Stack • Runnable Code Implementation (JAVASCRIPT) • Hands • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Outbound HTTP Timeouts • Production Hardening & Security: HttpOnly rotating JWTs, Zod boundary schemas, Helmet & OWASP BOLA guards

Purpose: To understand and apply Outbound HTTP Timeouts, Circuit Breakers & Exponential Backoff Retries in real Node js systems — specifically: Prevent one slow third-party paymen...

What Topic Is Next → Next Topic 15.3: Background Queue Workers, Dead-Letter Retries & Graceful SIGTERM Drain

15.3 Background Queue Workers, Dead-Letter Retries & Graceful SIGTERM Drain

[65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Background Queue Workers, Dead • Performance Optimization, Complexity & Latency Impact of Background Queue Workers, Dead • Live Debugging, Stack • Runnable Code Implementation (JAVASCRIPT) • Hands • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Background Queue Workers • Production Hardening & Security: HttpOnly rotating JWTs, Zod boundary schemas, Helmet & OWASP BOLA guards

Purpose: To understand and apply Background Queue Workers, Dead-Letter Retries & Graceful SIGTERM Drain in real Node js systems — specifically: Finish in-flight jobs safely during...

What Topic Is Next → Module 15 Capstone (Fault-Tolerant Webhook & Background Job Worker Service in Node.js) → Next Topic 16.1: Preventing Prototype Pollution, NoSQL/SQL Injection, SSRF & Token Replay

Module 15 Hands-On Capstone Project (100 Marks): Fault-Tolerant Webhook & Background Job Worker Service in Node.js — Architect, implement, test, and document 'Fault-Tolerant Webhook & Background Job Worker Service in Node.js' applying every bullet-wise engineering concept from Module 15 (Resilient Node.js...

MODULE 16 OF 16 • STAGE 4 • 14 LIVE + LAB HOURS • 3 TOPICS (24 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Node.js Security Hardening, Integration Testing & Backend System Design

Self-explanatory list-and-bullet production mastery module covering Node.js Security Hardening, Integration Testing & Backend System Design in Node js Training, paired with hands-on labs, capstone repository, Topic Mock Tests, and Spoken MockAttempt AI Viva.

Objectives: ✓ Preventing Prototype Pollution, NoSQL/SQL Injection, SSRF & Token Replay — Purpose: Lock down public Node.js APIs against the most common real-world attacker exploits. • ✓ Automated API Integration Testing with Node Test Runner / Vitest & Supertest — Purpose: Verify status codes, auth scopes, and database side effects automatically before deployment. • ✓ Event-Loop Latency Diagnostics, Memory Leaks & Node.js Interview Scenarios — Purpose: Spot blocked event loops, unbounded caches, and concurrency bottlenecks in live interviews. • ✓ Build, verify, and document 'OWASP-Hardened Express API Suite

with Supertest Automated Verification' with clean Git commits and edge-case protection. • ✓ Clear 100% of Topic Timed Mock Tests and Spoken Voice + MockAttempt AI Technical Interviews.

16.1 Preventing Prototype Pollution, NoSQL/SQL Injection, SSRF & Token Replay [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Preventing Prototype Pollution, NoSQL/SQL Injection, SSRF & Token Replay • Performance Optimization, Complexity & Latency Impact of Preventing Prototype Pollution, NoSQL/SQL Injection, SSRF.... • Live Debugging, Stack • Runnable Code Implementation (JAVASCRIPT) • Hands • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Preventing Prototype Pollution • Production Hardening & Security: HttpOnly rotating JWTs, Zod boundary schemas, Helmet & OWASP BOLA guards

Purpose: To understand and apply Preventing Prototype Pollution, NoSQL/SQL Injection, SSRF & Token Replay in real Node js systems — specifically: Lock down public Node.js APIs aga...

What Topic Is Next → Next Topic 16.2: Automated API Integration Testing with Node Test Runner / Vitest & Supertest

16.2 Automated API Integration Testing with Node Test Runner / Vitest & Supertest [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Automated API Integration Testing with Node Test Runner / Vitest & Superte... • Performance Optimization, Complexity & Latency Impact of Automated API Integration Testing with Node Test Runner /.... • Live Debugging, Stack • Runnable Code Implementation (JAVASCRIPT) • Hands • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Automated API Integration Testing with Node Test Run... • Production Hardening & Security: HttpOnly rotating JWTs, Zod boundary schemas, Helmet & OWASP BOLA guards

Purpose: To understand and apply Automated API Integration Testing with Node Test Runner / Vitest & Supertest in real Node js systems — specifically: Verify status codes, auth sco...

What Topic Is Next → Next Topic 16.3: Event-Loop Latency Diagnostics, Memory Leaks & Node.js Interview Scenarios

16.3 Event-Loop Latency Diagnostics, Memory Leaks & Node.js Interview Scenarios [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Event • Performance Optimization, Complexity & Latency Impact of Event • Live Debugging, Stack • Runnable Code Implementation (JAVASCRIPT) • Hands • Idiomatic NODE Implementation: strict TypeScript contracts, non-blocking I/O, idempotency keys & Redis cachin... • Node.js 22 V8 + Libuv Event Loop, Streams & React/Next.js Full-Stack Execution & Memory Lifecycle for Event-Loop Latency Diagnostics • Production Hardening & Security: HttpOnly rotating JWTs, Zod boundary schemas, Helmet & OWASP BOLA guards

Purpose: To understand and apply Event-Loop Latency Diagnostics, Memory Leaks & Node.js Interview Scenarios in real Node js systems — specifically: Spot blocked event loops, unbou...

What Topic Is Next → Module 16 Capstone → End-of-Course Live Client Project: SwiftDispatch High-Throughput Logistics API Gateway & Webhook Processor

Module 16 Hands-On Capstone Project (100 Marks): OWASP-Hardened Express API Suite with Supertest Automated Verification — Architect, implement, test, and document 'OWASP-Hardened Express API Suite with Supertest Automated Verification' applying every bullet-wise engineering concept from Module 16 (Node.js Secur...

3. LIVE CLIENT PRODUCTION PROJECT & CAREER SCOPE

End-of-Course Live Client Production Project + Official Internship Confirmation Letter

Project: SwiftDispatch High-Throughput Logistics API Gateway & Webhook Processor • **Confirmed Role:** Node.js Backend API Engineering Intern

Domain & Production Infra: Supply Chain & Hyperlocal Delivery Client (Node.js + Express + Cluster Workers + PM2/Docker HTTPS Production Server)

- Engineered non-blocking Node.js & Express REST APIs with modular controllers, request validation, and centralized error middleware.
- Implemented secure JWT authentication, RBAC permissions, HMAC webhook signature verification, and rate limiting.
- Built asynchronous background job queues, streaming file processors, and connection-pooled database repositories.
- Deployed hardened Node.js services to live client production with structured JSON logging and uptime health checks.

Target Job Roles: Node.js Backend Developer, REST API & Microservices Engineer, Backend SDE-1 (JavaScript/Node), Server-Side Application Engineer

Realistic Annual CTC Scope: **Fresher / Entry-Level (0–1 Yr):** ₹5.0 LPA – ₹9.5 LPA **Mid-Level Node.js Backend Engineer (1–3 Yrs):** ₹10.0 LPA – ₹20.0 LPA **Senior Backend / Systems Architect (3–6+ Yrs):** ₹20.0 LPA – ₹38.0+ LPA