📘 Node.js Interview Questions & Answers (Basic → Advanced)

🌱 BASIC LEVEL

1. What is Node.js?

👉 **Answer:**

- Node.js is a **runtime environment** built on **Google's V8 JavaScript engine**.
- It allows you to run **JavaScript on the server-side**.
- It uses an **event-driven, non-blocking I/O model**, making it efficient and scalable.

✅ Example:

```
console.log("Hello from Node.js!");
```

---

2. What is the difference between Node.js and JavaScript in the browser?

👉 **Answer:**

- **Browser JS** → Used for frontend, interacts with DOM, has APIs like window, document.
- **Node.js** → Used for backend, has APIs like fs, http, process.

---

3. What are modules in Node.js?

👉 **Answer:**
Modules are reusable pieces of code. Node.js has:

- **Core modules** → (fs, http, path, etc.)
- **Local modules** → User-defined
- **Third-party modules** → Installed via NPM (express, mongoose)

✅ Example:

```
const fs = require("fs"); // Core module

fs.writeFileSync("demo.txt", "Hello Node!");
```

---

4. Difference between require() and import

👉 **Answer:**

- require() → CommonJS, default in Node.js.
- import → ES6 modules, needs "type": "module" in package.json.

✅ Example:

```
// CommonJS
const http = require("http");

// ES6
import http from "http";
```

---

5. What is npm?

👉 **Answer:**

- NPM = Node Package Manager
- Used to **install, share, and manage dependencies**.

✅ Example:

```
npm init -y   # Initialize project
npm install express
```

⚡ **INTERMEDIATE LEVEL**

6. Explain Event Loop in Node.js

👉 **Answer:**

- Node.js is **single-threaded** but handles concurrency using the **event loop**.
- It processes requests in **phases**: timers, I/O callbacks, idle/prepare, poll, check, close callbacks.
- Uses **libuv** library internally.

✅ Example:

```
console.log("Start");

setTimeout(() => {
  console.log("Inside Timeout");
}, 0);

console.log("End");
```

```
// Output:
// Start
// End
// Inside Timeout
```

---

7. Difference between process.nextTick() and setImmediate()

👉 **Answer:**

- process.nextTick() → Executes **before** event loop continues.
- setImmediate() → Executes **after** the poll phase in event loop.

---

8. What are Streams in Node.js?

👉 **Answer:**
Streams process data **chunk by chunk** instead of loading everything at once.

Types:

- **Readable** → Read data
- **Writable** → Write data
- **Duplex** → Both
- **Transform** → Modify while reading/writing

✅ Example:

```
const fs = require("fs");
const readStream = fs.createReadStream("input.txt");
readStream.on("data", chunk => {
  console.log("Received:", chunk.toString());
});
```

---

9. Explain Middleware in Express

👉 **Answer:**
Middleware is a function that **executes before route handler**.
It can:

- Modify req or res
- End request/response cycle
- Call next middleware

✅ Example:

```
const express = require("express");
const app = express();

// Middleware
app.use((req, res, next) => {
  console.log("Request URL:", req.url);
  next();
});

app.get("/", (req, res) => res.send("Hello World"));
app.listen(3000);
```

---

10. Difference between res.send(), res.json(), and res.end()

👉 **Answer:**

- res.send() → Sends string/object/HTML.
- res.json() → Sends JSON response.
- res.end() → Ends response without data.

---

11. Explain Mongoose Schema and Model

👉 **Answer:**

- **Schema** → Defines structure of MongoDB document.
- **Model** → Wrapper around schema, used for CRUD operations.

✅ Example:

```
const mongoose = require("mongoose");

const userSchema = new mongoose.Schema({
  name: String,
  email: String
});
```

```
const User = mongoose.model("User", userSchema);
```

---

12. What is JWT Authentication?

👉 **Answer:**

- **JWT (JSON Web Token)** is used for secure authentication.
- User logs in → server returns token → client stores token → sends in headers for protected routes.

✅ Example:

```
const jwt = require("jsonwebtoken");

const token = jwt.sign({ id: 1 }, "secret", { expiresIn: "1h" });
console.log(token);
```

🚀 **ADVANCED LEVEL**

13. What is Clustering in Node.js?

👉 **Answer:**

- Node.js is single-threaded.
- **Cluster module** allows running multiple worker processes on different CPU cores.

✅ Example:

```
const cluster = require("cluster");
const http = require("http");
const os = require("os");

if (cluster.isMaster) {
 for (let i = 0; i < os.cpus().length; i++) {
  cluster.fork();
 }
} else {
 http.createServer((req, res) => {
  res.end("Handled by worker " + process.pid);
  }).listen(3000);
}
```

---

14. Difference between Cluster and Worker Threads

👉 **Answer:**

- **Cluster** → Creates multiple Node.js processes (workers).
- **Worker Threads** → Run JavaScript in parallel within the same process (lighter).

---

15. How to Handle Uncaught Errors in Node.js?

👉 **Answer:**

```
process.on("uncaughtException", (err) => {
  console.error("Uncaught Error:", err);
});
process.on("unhandledRejection", (reason) => {
  console.error("Unhandled Rejection:", reason);
});
```

---

16. How do you improve performance in Node.js apps?

👉 **Answer:**

- Use **clustering / load balancing**
- Use **caching (Redis)**
- Optimize queries (MongoDB indexes)
- Use **streams** for large files
- Avoid blocking code

---

17. What is the difference between synchronous blocking vs asynchronous non-blocking in Node.js?

👉 **Answer:**

- **Blocking (sync)** → Waits until operation finishes.
- **Non-blocking (async)** → Continues executing other tasks while waiting.

✅ Example:

```
// Blocking
const data = fs.readFileSync("file.txt");
console.log(data);

// Non-blocking
fs.readFile("file.txt", (err, data) => console.log(data));
```

---

18. How to implement Rate Limiting in Express?

👉 **Answer:**
Using express-rate-limit:

```
const rateLimit = require("express-rate-limit");

const limiter = rateLimit({
  windowMs: 15 * 60 * 1000,
  max: 100
});

app.use(limiter);
```

---

19. How to implement Caching in Node.js?

👉 **Answer:**
Using **Redis** for caching database queries.

```
const redis = require("redis");
const client = redis.createClient();

app.get("/data", async (req, res) => {
  client.get("key", async (err, data) => {
    if (data) return res.send(data);
    const freshData = await getFromDB();
    client.set("key", JSON.stringify(freshData));
    res.send(freshData);
  });
```

});

---

20. How to Debug Node.js Apps?

👉 **Answer:**

- Use console.log() (basic)
- Use **Node Inspector** (node --inspect app.js)
- Debug via **VS Code debugger**
- Use logging libraries (winston, morgan)

1. What is the difference between Node.js and Python for backend development?

👉 **Answer:**

- **Node.js** → Non-blocking, event-driven, best for real-time apps.
- **Python** → Multi-threaded, better for CPU-heavy apps (ML, Data Science).

---

2. What is the difference between fs.readFile and fs.createReadStream?

👉 **Answer:**

- fs.readFile → Reads entire file into memory.
- fs.createReadStream → Reads file **chunk by chunk** (better for large files).

---

3. What is the difference between exports and module.exports in Node.js?

👉 **Answer:**

- exports → Shortcut to module.exports.
- Only module.exports is returned by require().

✅ Example:

```
// file1.js
module.exports = { a: 10 };

// file2.js
const x = require("./file1");
console.log(x.a); // 10
```

---

4. What is REPL in Node.js?

👉 **Answer:**
REPL = **Read, Eval, Print, Loop**. It is Node's interactive shell.

```
node
> 2+3
5
```

---

5. What is the difference between __dirname and process.cwd()?

👉 **Answer:**

- __dirname → Directory of current file.
- process.cwd() → Directory where Node process started.

⚡ **INTERMEDIATE QUESTIONS**

6. What is the difference between callback hell and async/await?

👉 **Answer:**

- **Callback hell** → Nested callbacks (hard to read).
- **Async/await** → Cleaner, synchronous-looking code.

✅ Example:

```
// Callback hell
fs.readFile("a.txt", () => {
  fs.readFile("b.txt", () => {
    fs.readFile("c.txt", () => {});
  });
});

// Async/await
const read = util.promisify(fs.readFile);
async function run() {
  const a = await read("a.txt");
  const b = await read("b.txt");
}
```

---

7. What are Environment Variables in Node.js?

👉 **Answer:**
Environment variables store **config/secrets** outside code.

✅ Example:

```
PORT=4000

console.log(process.env.PORT); // 4000
```

---

8. What is the difference between PUT and PATCH in REST API?

👉 **Answer:**

- **PUT** → Replaces the entire resource.
- **PATCH** → Updates only part of the resource.

---

9. How does Node.js handle multiple requests with a single thread?

👉 **Answer:**

- Uses **event loop + callback queue**.
- Heavy I/O handled asynchronously (non-blocking).
- Offloads work to **libuv thread pool**.

---

10. What is the difference between cookie-based and token-based authentication?

👉 **Answer:**

- **Cookie-based** → Session stored in server.
- **Token-based (JWT)** → Token stored in client, stateless server.

🚀 ADVANCED QUESTIONS

11. What is the difference between process and thread in Node.js?

👉 **Answer:**

- **Process** → Instance of program with memory.
- **Thread** → Smallest execution unit inside process.
- Node.js = single process, event-loop + background threads (libuv).

---

12. What is the difference between Monolithic and Microservices architecture?

👉 **Answer:**

- **Monolithic** → One big codebase.
- **Microservices** → Multiple small services communicating via APIs.

---

13. What are Worker Threads in Node.js?

👉 **Answer:**

- Allow running JS in parallel threads.
- Useful for **CPU-heavy tasks**.

✅ Example:

```
const { Worker } = require("worker_threads");
new Worker("./worker.js", { workerData: { num: 5 } });
```

---

14. How do you secure a Node.js API?

👉 **Answer:**

- Use **Helmet.js** for headers.
- Use **rate limiting** (express-rate-limit).
- Use **JWT** or OAuth2.
- Avoid **eval()**.
- Sanitize inputs.

---

15. What are WebSockets in Node.js?

👉 **Answer:**

- WebSockets enable **real-time communication** (chat, notifications).
- Unlike HTTP, it keeps **persistent connection**.

✅ Example:

```
const WebSocket = require("ws");
const server = new WebSocket.Server({ port: 8080 });

server.on("connection", ws => {
  ws.on("message", msg => console.log(msg));

  ws.send("Hello Client!");

});
```

---

16. What is the difference between PM2 and nodemon?

👉 **Answer:**

- **nodemon** → Restarts app on file changes (development).
- **PM2** → Production process manager (load balancing, monitoring, clustering).

---

17. How do you handle file uploads in Node.js?

👉 **Answer:**
Using multer:

```
const multer = require("multer");

const upload = multer({ dest: "uploads/" });


app.post("/upload", upload.single("file"), (req, res) => {

  res.send("File uploaded");

});
```

---

18. How to connect Node.js with MongoDB?

👉 **Answer:**

```
const mongoose = require("mongoose");


mongoose.connect("mongodb://localhost:27017/test", {

  useNewUrlParser: true,

  useUnifiedTopology: true

}).then(() => console.log("Connected"));
```

---

19. What is the difference between GraphQL and REST?

👉 **Answer:**

- **REST** → Fixed endpoints, returns full data.

- **GraphQL** → Single endpoint, client requests only required data.

---

20. How do you scale a Node.js application?

👉 **Answer:**

- Use **Clustering** (multi-core CPUs).
- Use **Load Balancer** (NGINX, HAProxy).
- Use **Microservices**.
- Use **Redis caching**.
- Use **Docker/Kubernetes**.

1. Why do we prefer Promises/Async-Await over Callbacks in Node.js?

👉 **Answer:**

- **Callbacks** → Lead to **callback hell** (nested, unreadable code). Error handling is harder.
- **Promises** → Provide **chaining** and better error handling (.catch()).
- **Async/Await** → Makes async code look synchronous → more **readable & maintainable**.

✅ Example:

Callback hell ⛔

```
fs.readFile("a.txt", (err, dataA) => {
  fs.readFile("b.txt", (err, dataB) => {
    console.log(dataA, dataB);
  });
});
```

Promise + Async/Await ✅

```
const readFile = util.promisify(fs.readFile);

async function run() {
  const dataA = await readFile("a.txt", "utf-8");
  const dataB = await readFile("b.txt", "utf-8");
  console.log(dataA, dataB);
}
run();
```

---

2. Why is Node.js single-threaded, and how does it still handle many requests?

👉 **Answer:**

- Node.js is **single-threaded** to simplify concurrency (no deadlocks).
- It uses **event loop + libuv thread pool** → Offloads heavy I/O to worker threads.
- This makes it **non-blocking** and suitable for high-concurrency apps.

---

3. Why use process.nextTick() instead of setImmediate()?

👉 **Answer:**

- process.nextTick() runs **before the event loop continues**.
- Useful when you want to execute something **immediately after the current function**.
- setImmediate() waits for the **next cycle** → useful when you want I/O first.

---

4. Why use Middleware in Express instead of writing everything in routes?

👉 **Answer:**

- Middleware promotes **reusability & clean code**.
- Helps in:
  - Logging requests
  - Authentication
  - Error handling
- Without middleware → code duplication in every route.

---

5. Why use res.json() instead of res.send() in Express?

👉 **Answer:**

- res.send() → Sends any type (string, HTML, buffer, object).
- res.json() → Specifically converts object → JSON string.
- For APIs, res.json() is **safer & more explicit**.

---

6. Why use JWT instead of Session-based Authentication?

👉 **Answer:**

- **Session-based**: Stores session on **server memory/DB** → not scalable.
- **JWT (Token-based)**: Encoded token stored in **client** → server remains **stateless**, scalable.

- Best for **microservices & mobile apps**.

---

7. Why use Streams instead of reading full files in Node.js?

👉 **Answer:**

- fs.readFile() loads **entire file into memory** → bad for large files.
- Streams process **chunk by chunk** → efficient, faster, memory-friendly.

✅ Example:

```
const stream = fs.createReadStream("bigfile.txt");

stream.on("data", chunk => console.log("Received chunk:", chunk.length));
```

---

8. Why use async/await instead of .then() with Promises?

👉 **Answer:**

- .then() → Works but can get messy with multiple chains.
- async/await → Cleaner, readable, easier debugging (like synchronous code).

---

9. Why use Clustering in Node.js?

👉 **Answer:**

- Node.js runs on a **single CPU core**.
- **Cluster module** allows us to run multiple processes → utilize all cores.
- Increases **performance & scalability**.

---

10. Why use Redis caching in Node.js apps?

👉 **Answer:**

- Database queries are slow compared to in-memory lookup.
- Redis stores **frequently used data in memory**.
- Reduces **latency** & improves **API speed**.

---

11. Why use GraphQL over REST in Node.js?

👉 **Answer:**

- **REST** → Multiple endpoints, fixed response structure.
- **GraphQL** → Single endpoint, client decides what data it needs.

- Avoids **over-fetching / under-fetching**.

---

12. Why use PM2 instead of Nodemon in production?

👉 **Answer:**

- **Nodemon** → Development only (auto restart on file change).
- **PM2** → Production process manager with:
  - Load balancing
  - Monitoring
  - Log management
  - Cluster support

---

13. Why use helmet in Express?

👉 **Answer:**

- helmet secures Express apps by setting HTTP headers.
- Protects against XSS, clickjacking, MIME sniffing.

---

14. Why use async_hooks in Node.js?

👉 **Answer:**

- To **track async operations** across the event loop.
- Useful for debugging, profiling, monitoring.

---

15. Why use Worker Threads instead of Cluster in Node.js?

👉 **Answer:**

- **Cluster** → Creates multiple processes (good for I/O).
- **Worker Threads** → Parallel execution in **same process** (good for CPU-heavy tasks).

1. Why not use callbacks anymore?

- **Problem with Callbacks (Callback Hell):**
  - Nested callbacks make code hard to read and maintain.
  - Example:
  - getUser(id, function(user) {
  -   getPosts(user.id, function(posts) {

    - getComments(posts[0].id, function(comments) {
    - console.log(comments);
    - });
    - });
    - });

- **Solution with Promises:**
- getUser(id)
- .then(user => getPosts(user.id))
- .then(posts => getComments(posts[0].id))
- .then(comments => console.log(comments))
- .catch(err => console.error(err));
- **Solution with Async/Await:**
- async function fetchData() {
- try {
- const user = await getUser(id);
- const posts = await getPosts(user.id);
- const comments = await getComments(posts[0].id);
- console.log(comments);
- } catch (err) {
- console.error(err);
- }
- }
- fetchData();

👉 So today, **callbacks are rare** — Promises and async/await are preferred for readability.

---

2. What is EventEmitter in Node.js?

- Node.js has an **event-driven architecture**.
- The events module provides EventEmitter.
- Example:
- const EventEmitter = require('events');
- const emitter = new EventEmitter();

-
- // listener
- emitter.on('greet', name => {
-   console.log(`Hello, ${name}`);
- });
-
- // emit
- emitter.emit('greet', 'Darshan');

👉 Useful in building chat apps, streams, socket servers.

---

3. What are Streams in Node.js?

- Streams handle **large data** efficiently (instead of loading whole file into memory).
- Types:
  - Readable (e.g., fs.createReadStream)
  - Writable (e.g., fs.createWriteStream)
  - Duplex (both)
  - Transform (data modification)

Example:

```
const fs = require('fs');
const read = fs.createReadStream('input.txt');
const write = fs.createWriteStream('output.txt');

read.pipe(write); // transfer data chunk by chunk
```

👉 Saves memory & improves performance.

---

4. What is Cluster in Node.js?

- Node.js is **single-threaded** but can use multiple cores.
- cluster module allows running multiple Node processes.

```
const cluster = require('cluster');
const http = require('http');
const os = require('os');
```

```
if (cluster.isMaster) {
  const numCPUs = os.cpus().length;
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }
} else {
  http.createServer((req, res) => {
    res.end('Handled by worker ' + process.pid);
  }).listen(3000);
}
```

👉 Increases performance on multi-core machines.

---

5. Difference between CommonJS and ES Modules

- **CommonJS (require)**
  - Default in Node.js before ES6.
  - Loads modules synchronously.
  - Example:
  - const fs = require('fs');
- **ESM (import/export)**
  - Modern JavaScript standard.
  - Loads asynchronously.
  - Example:
  - import fs from 'fs';

👉 Today, both are supported, but **ESM is future-proof**.

---

6. What is Middleware in Express.js?

- Middleware is a function that has access to req, res, next.
- Example:
- app.use((req, res, next) => {
-   console.log(`Request: ${req.method} ${req.url}`);

- next();
- });

👉 Used for authentication, logging, error handling.

---

7. How to handle security in Node.js?

- Use **Helmet.js** for securing HTTP headers.
- Use **dotenv** for environment variables.
- Prevent **SQL Injection** (use parameterized queries).
- Prevent **XSS** (sanitize input).
- Always **hash passwords** (bcrypt).

---

8. Difference between Process and Thread in Node.js

- **Process**: Independent execution with own memory.
- **Thread**: Lightweight execution inside a process.
- Node.js runs in **single-threaded event loop**, but can spawn multiple processes (cluster) or use worker threads.

---

9. What are Worker Threads?

- Introduced in Node.js v10.5.0.
- Allow running **JavaScript in parallel threads** (CPU-intensive tasks).

Example:

```
const { Worker } = require('worker_threads');


new Worker(`
  const { parentPort } = require('worker_threads');
  parentPort.postMessage('Hello from Worker!');
`, { eval: true });
```

---

10. When to use Redis in Node.js?

- Redis is used for:
  - Caching results (reduce DB hits).

- Session storage.
- Pub/Sub system (chat apps).

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

Answer:

These all schedule asynchronous code execution, but they run in different phases of the **event loop**:

1. **process.nextTick()**
    - Executes **immediately after the current operation**, before the event loop continues.
    - Priority: **Highest**
    - Example:

```
console.log("Start");

process.nextTick(() => {
  console.log("Next Tick");
});

console.log("End");
```

✅ Output:

```
Start

End

Next Tick
```

2. **setImmediate()**
    - Executes **in the check phase** of the event loop (after I/O).
    - Runs **after pending I/O** events are processed.
3. **setTimeout(fn, 0)**
    - Executes **in the timers phase**.
    - Not guaranteed to run immediately—it depends on the event loop state.

👉 **Key difference:**

- process.nextTick() → Highest priority (runs before anything else).
- setImmediate() → Runs after I/O callbacks.
- setTimeout(fn, 0) → Scheduled for next cycle (timers phase).

🔷 2. What is middleware in Node.js (Express)? How does it work?

Answer:

Middleware are **functions** that have access to the req, res, and next objects in Express.js.
They allow us to execute code, modify request/response objects, and control the request flow.

```
const express = require("express");

const app = express();


// Custom middleware

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

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

  next(); // Pass control to next middleware/route

});


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

  res.send("Hello, Middleware!");

});


app.listen(3000, () => console.log("Server running on port 3000"));
```

✅ Middleware can be:

- **Application-level** (like above).
- **Router-level** (express.Router()).
- **Built-in** (like express.json()).
- **Error-handling middleware** (takes 4 args: err, req, res, next).

🔷 3. What is CORS in Node.js? How do you handle it?

Answer:

CORS (Cross-Origin Resource Sharing) allows a web app running on one domain (e.g., http://localhost:3000) to request resources from another domain (e.g., http://api.example.com).

Without CORS, browsers **block cross-origin requests** for security reasons.

Solution in Express:

```
const express = require("express");
const cors = require("cors");

const app = express();
app.use(cors()); // Enable CORS for all routes

app.get("/", (req, res) => {
  res.json({ message: "CORS enabled!" });
});

app.listen(4000, () => console.log("Server running on 4000"));
```

✅ You can also configure specific domains:

```
app.use(cors({ origin: "http://localhost:3000" }));
```

🔷 4. What is difference between CommonJS and ES Modules in Node.js?

Answer:

1. **CommonJS (CJS)**
   - Uses require() and module.exports.
   - Default in older Node.js versions.
   - Example:
   - const fs = require("fs");
   - module.exports = { myFunc };
2. **ES Modules (ESM)**
   - Uses import and export.
   - Default in modern Node.js (>= v14 with "type": "module" in package.json).
   - Example:
   - import fs from "fs";
   - export default myFunc;

👉 **Difference:**

- CJS = **synchronous** (good for server-side).
- ESM = **asynchronous** (better for modern apps, tree-shaking, etc.).

🔷 5. What are Worker Threads in Node.js?

Answer:

Node.js is **single-threaded**, but sometimes CPU-intensive tasks (like image processing, encryption) can block the event loop.
Worker Threads allow running **JavaScript in parallel threads**.

```
const { Worker } = require("worker_threads");


const worker = new Worker(`
  const { parentPort } = require('worker_threads');
  let result = 0;
  for (let i = 0; i < 1e9; i++) result += i;
  parentPort.postMessage(result);
`, { eval: true });


worker.on("message", (msg) => console.log("Result:", msg));
```

✅ Useful for **CPU-bound tasks**.
For **I/O-bound tasks**, event loop is sufficient.

🔷 6. What is JWT (JSON Web Token) and how do you use it in Node.js?

Answer:

JWT is a **token-based authentication** mechanism.

- **Header** → Algorithm & token type.
- **Payload** → User info (claims).
- **Signature** → Ensures integrity.

Example with jsonwebtoken:

```
const jwt = require("jsonwebtoken");


const token = jwt.sign({ userId: 123 }, "secretKey", { expiresIn: "1h" });
console.log("JWT:", token);


const decoded = jwt.verify(token, "secretKey");
```

console.log("Decoded:", decoded);

✅ Used in login systems for **stateless authentication**.

🔷 7. Difference between Monolithic and Microservices architecture in Node.js?

Answer:

- **Monolithic**
  - Single codebase.
  - Easier to build, harder to scale.
  - Example: A single Express app serving API, UI, DB access.
- **Microservices**
  - Multiple independent services (auth, payment, order).
  - Communicate via APIs or message brokers.
  - Scalable, fault-tolerant, but more complex.

---

1. Why use Promises instead of Callbacks?

Answer:

- **Callbacks**: Functions passed as arguments to handle results/errors asynchronously.
  - Example:
  - fs.readFile("file.txt", (err, data) => {
  -   if (err) console.error(err);
  -   else console.log(data.toString());
  - });
  - Problem: Leads to **callback hell** when multiple async calls are nested.
- **Promises**: Provide a cleaner way to handle async logic.
  - Example:
  - fs.promises.readFile("file.txt", "utf-8")
  -   .then(data => console.log(data))
  -   .catch(err => console.error(err));
- **Advantages:**
  - Avoids **callback hell**.
  - Better **error handling** with .catch().

    - Can use async/await for synchronous-looking code.

---

2. What is async/await and how is it better than Promises?

Answer:

- async/await is **syntactic sugar** over Promises.
- Example with Promise:
- fetchData()
-   .then(data => process(data))
-   .catch(err => console.error(err));
- Same with async/await:
- async function getData() {
-   try {
-     const data = await fetchData();
-     console.log(data);
-   } catch (err) {
-     console.error(err);
-   }
- }
- getData();
- **Why better?**
    - Looks synchronous.
    - Easier to debug.
    - Reduces .then().catch() nesting.

---

3. Difference between process.nextTick(), setImmediate(), and setTimeout()?

Answer:

- process.nextTick() → Executes **immediately after current operation**, before next event loop tick.
- setImmediate() → Executes in the **check phase** of event loop (after I/O).
- setTimeout(fn, 0) → Executes after **minimum 1ms delay** in the **timers phase**.

Example:

```
console.log("Start");


process.nextTick(() => console.log("nextTick"));
setImmediate(() => console.log("setImmediate"));
setTimeout(() => console.log("setTimeout"), 0);


console.log("End");
```

Output Order:

```
Start
End
nextTick
setTimeout
setImmediate
```

---

4. What is the difference between CommonJS (require) and ES Modules (import)?

Answer:

- **CommonJS (CJS):**
    - Uses require().
    - Synchronous.
    - Default in Node.js before v13.
- const fs = require("fs");
- **ES Modules (ESM):**
    - Uses import/export.
    - Asynchronous & modern.
    - Default in modern Node.js.
- import fs from "fs";
- **Key Difference:** CJS executes immediately, ESM allows **tree-shaking** (better optimization).

---

5. What is clustering in Node.js and why use it?

Answer:

- Node.js runs in a **single thread**, so it cannot use all CPU cores by default.

- **Clustering** creates **multiple worker processes** that share the same server port.
- Example:
- const cluster = require("cluster");
- const http = require("http");
- const os = require("os");
-
- if (cluster.isMaster) {
-   const numCPUs = os.cpus().length;
-   for (let i = 0; i < numCPUs; i++) cluster.fork();
- } else {
-   http.createServer((req, res) => {
-     res.writeHead(200);
-     res.end("Hello from worker " + process.pid);
-   }).listen(3000);
- }
- **Why?**
  - Increases performance by using all CPU cores.
  - Better scalability.

---

6. What are Worker Threads in Node.js?

Answer:

- **Worker Threads** allow running JavaScript code in **parallel threads** (not just event loop).
- Useful for **CPU-heavy tasks** (e.g., image processing, ML models).
- Example:
- const { Worker } = require("worker_threads");
-
- const worker = new Worker(`
-   const { parentPort } = require("worker_threads");
-   parentPort.postMessage("Hello from worker!");
- `, { eval: true });
-

- worker.on("message", msg => console.log(msg));
- **Difference from Cluster:**
  - Cluster → Multiple Node.js processes (multi-core scaling).
  - Worker Threads → Parallel execution inside a single Node.js process.

---

7. Explain Streams in Node.js. Why use them?

Answer:

- **Streams** handle data **chunk by chunk** instead of loading all into memory.
- Types:
  - **Readable**: Read from source (e.g., file).
  - **Writable**: Write to destination (e.g., file).
  - **Duplex**: Both read & write.
  - **Transform**: Modify data while streaming.

Example:

```
const fs = require("fs");


const readStream = fs.createReadStream("input.txt");
const writeStream = fs.createWriteStream("output.txt");


readStream.pipe(writeStream);
```

Why use Streams?

- Efficient memory usage.
- Faster data processing.
- Best for large files / video streaming.

---

8. What is Middleware in Express?

Answer:

- Middleware = Functions that run **before request reaches route handler**.
- Example:
- app.use((req, res, next) => {
-   console.log("Request Time:", Date.now());

-   next();
- });
- Types:
  - **Application-level**
  - **Router-level**
  - **Error-handling**
  - **Built-in (e.g., express.json())**
  - **Third-party (e.g., morgan, cors)**