150+ Node.js Interview Questions Answers

Follow :https://www.linkedin.com/in/sandeeppal/

Node.js Basics (Beginner to Intermediate)

1. What is Node.js?

Node.js is an open-source, cross-platform runtime environment that allows you to run JavaScript on the server side. It's built on the V8 JavaScript engine developed by Google (the same one used in Chrome). With Node.js, you can build scalable and high-performance web applications, APIs, and even real-time services like chat apps.

📌 **Example:**
If you write a simple HTTP server in Node.js, you can serve a webpage without using a traditional web server like Apache:

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

http.createServer((req, res) => {
  res.end('Hello from Node.js server!');
}).listen(3000);
```

---

2. How does Node.js work?

Node.js operates on a **single-threaded, event-driven** architecture. It uses the **event loop** to handle multiple connections concurrently, which means it can perform non-blocking I/O operations efficiently.

📌 **Real-life analogy:**
Think of it like a chef (Node.js) who takes multiple orders (requests) but doesn't cook each dish one at a time. Instead, they prep and send off tasks (e.g., grilling, baking) and move on to the next customer while waiting.

---

3. What is the V8 engine?

The V8 engine is a high-performance JavaScript engine developed by Google for Chrome. Node.js uses it to compile and run JavaScript code on the server side. It converts JS code into machine code, making it fast and efficient.

📌 **Use Case:**
When you run a `.js` file with Node.js, the V8 engine compiles your code into machine code behind the scenes.

---

4. What are the key features of Node.js?

- **Asynchronous and Event-Driven:** Handles multiple requests without blocking.
- **Fast Execution:** Powered by the V8 engine.
- **Single-Threaded but Scalable:** Uses event loop and callbacks for handling concurrency.
- **Cross-platform:** Runs on Windows, Linux, and macOS.
- **NPM (Node Package Manager):** Massive ecosystem of reusable packages.

---

5. How is Node.js different from traditional web servers like Apache?

Feature	Node.js	Apache
Thread Model	Single-threaded event loop	Multi-threaded
I/O	Non-blocking	Blocking by default
Performance	Very high for I/O operations	Good but resource intensive
Use Case	Real-time apps, APIs	Websites, PHP apps

📌 **Example:**
For a chat application or API server with thousands of concurrent users, Node.js performs better than Apache.

---

6. What is the event loop in Node.js?

The event loop is the core of Node.js's non-blocking I/O operations. It handles all asynchronous operations in a single thread, constantly checking for events (e.g., incoming data, timers) and executing associated callbacks.

📌 **Example:**

```
setTimeout(() => {
  console.log('Executed after 2 seconds');
}, 2000);
```

This callback is scheduled by the event loop and executed when the time is up.

---

7. Explain the non-blocking I/O model in Node.js.

Non-blocking I/O means the application doesn't wait for a task (like reading a file or querying a database) to complete before moving to the next one. Node.js uses callbacks, promises, or async/await to handle the results when they're ready.

📌 **Example:**

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

fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});

console.log('Reading file...');
```

You'll see "Reading file..." first, even though the file is being read.

---

8. What are global objects in Node.js?

Global objects are available in all modules without the need to import them. Examples include:

- `__dirname`: Directory name of the current module
- `__filename`: Full path of the current module
- `global`: Similar to `window` in browsers
- `process`: Provides info about the current Node.js process
- `setTimeout`, `setInterval`, etc.

---

9. What is process in Node.js?

`process` is a global object that provides information and control over the current Node.js process.

📌 **Examples:**

```
console.log(process.pid); // Process ID
console.log(process.platform); // OS platform
```

You can also handle exit events:

```
process.on('exit', () => {
  console.log('Exiting...');
});
```

---

10. What is the use of `__dirname` and `__filename`?

- `__dirname`: Returns the directory path of the current module.
- `__filename`: Returns the full file path of the current module.

📌 **Example:**

```
console.log(__dirname);  // /Users/yourname/project
console.log(__filename); // /Users/yourname/project/app.js
```

These are very useful for reading or writing files relative to the script's location.

PM and Module Management

11. What is NPM?

NPM stands for **Node Package Manager**. It is the default package manager for Node.js and is used to install, share, and manage reusable packages or libraries.

It comes pre-installed with Node.js and gives you access to a huge ecosystem of open-source tools.

📌 **Example Use:**

```
npm install express
```

This command downloads the Express.js library into your project.

✅ NPM also:

- Manages package versions
- Handles dependencies
- Supports scripts to automate tasks (`npm run build`, `npm test`, etc.)

---

12. How do you install a package globally vs locally?

Local Installation (default):
Installs the package into the `node_modules` folder of your current project.

```
npm install lodash
```

- ✅ Used when the package is needed as part of your app's code.

Global Installation:
Installs the package system-wide, making it available in the command line anywhere.

```
npm install -g nodemon
```

- ✅ Used for tools/CLI apps like `nodemon`, `eslint`, `typescript`, etc.

---

13. What is the difference between dependencies and devDependencies?

- **dependencies**: These are required for your app to run in production.

  📌 Example: `express`, `mongoose`

- **devDependencies**: Only needed during development (testing, building, linting).

  📌 Example: `nodemon`, `eslint`, `jest`

📦 These are defined in `package.json`:

```
"dependencies": {
  "express": "^4.18.0"
},
"devDependencies": {
  "nodemon": "^3.0.0"
}
```

✅ Install a dev dependency with:

```
npm install nodemon --save-dev
```

---

14. What is semantic versioning?

Semantic Versioning (semver) is a versioning system that uses the format:

```
MAJOR.MINOR.PATCH
```

Example: `2.5.3`

- **MAJOR**: Breaking changes
- **MINOR**: New features, no breaking changes
- **PATCH**: Bug fixes, backwards-compatible

📌 **Example:**
If a package moves from `1.2.0` to `2.0.0`, it likely has breaking changes.

In `package.json`, you might see:

```
"express": "^4.17.1"
```

- `^` means it can auto-update minor and patch versions, but not major ones.

---

15. How do you update a package in Node.js?

To update all packages to their latest **safe versions** based on semver:

```
npm update
```

To update a specific package to its latest version:

```
npm install express@latest
```

For a more interactive way:

```
npx npm-check-updates -u
npm install
```

✅ The `npm-check-updates` tool helps update versions in your `package.json`.

---

16. How do you create a custom Node.js module?

You can create a custom module by writing logic in a separate `.js` file and exporting it using `module.exports`.

📌 **Example:**

mathUtils.js

```
function add(a, b) {
  return a + b;
}

function subtract(a, b) {
  return a - b;
}

module.exports = { add, subtract };
```

app.js

```
const math = require('./mathUtils');

console.log(math.add(5, 3));       // 8
console.log(math.subtract(9, 4));  // 5
```

✅ This is how you break your code into reusable pieces (modules) in Node.js.

REST API Development in Node.js

17. What is REST?

REST (Representational State Transfer) is an architectural style for designing networked applications. It uses standard HTTP methods to perform CRUD operations on resources, which are typically represented as URLs.

✅ A RESTful API allows different systems (like frontend apps or mobile apps) to interact with your server over HTTP in a stateless manner.

📌 **Example:**

- `GET /users` – Get all users
- `POST /users` – Create a new user
- `PUT /users/1` – Update user with ID 1
- `DELETE /users/1` – Delete user with ID 1

---

18. What are HTTP methods used in REST?

RESTful APIs use the following standard HTTP methods:

Method	Purpose
`GET`	Retrieve data
`POST`	Create new data
`PUT`	Update existing
`PATCH`	Partial update
`DELETE`	Remove data

---

19. What are status codes? Name commonly used ones.

HTTP status codes indicate the result of a request:

Code	Meaning
200	OK
201	Created
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not Found
500	Internal Server Error

📌 **Example:**

```
res.status(201).json({ message: 'User created successfully' });
```

---

20. How do you create a REST API using Node.js?

Using **Express.js**, you can quickly set up routes and logic.

📌 **Example:**

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

app.use(express.json());

app.get('/api/users', (req, res) => {
  res.json([{ id: 1, name: 'Alice' }]);
});

app.listen(3000, () => console.log('Server running'));
```

---

21. What is Express.js?

Express.js is a fast, minimal, and flexible Node.js web framework. It simplifies building web applications and APIs by handling routing, middleware, requests, responses, and more.

✅ It's like jQuery for the backend — removes the boilerplate.

---

22. How do you handle routing in Express?

Routing refers to defining how the application responds to different HTTP methods and URLs.

📌 **Example:**

```
app.get('/users', (req, res) => {
  res.send('Get all users');
});

app.post('/users', (req, res) => {
  res.send('Create user');
});
```

---

23. What are route parameters in Express?

Route parameters are dynamic values in the URL, defined using `:`.

📌 **Example:**

```
app.get('/users/:id', (req, res) => {
  const id = req.params.id;
  res.send(`User ID: ${id}`);
});
```

Request to `/users/42` returns: `User ID: 42`

---

24. How do you parse JSON request bodies in Express?

Use Express's built-in middleware:

```
app.use(express.json());
```

This enables the app to read `req.body` in JSON POST/PUT requests.

---

25. How do you handle CORS in Node.js?

CORS (Cross-Origin Resource Sharing) allows servers to specify who can access their APIs.

✅ Use the `cors` middleware:

```
npm install cors
```

```
const cors = require('cors');
app.use(cors());
```

You can also customize it:

```
app.use(cors({ origin: 'https://example.com' }));
```

---

26. What is middleware in Express.js?

Middleware is a function that runs **between** the request and response cycle. It can:

- Modify request/response
- Execute logic
- Call the next middleware

📌 **Example:**

```
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next(); // Move to next middleware or route
});
```

27. What is the difference between `app.use()` and `app.get()`?

Method	Purpose
`app.use()`	Middleware for **all requests**
`app.get()`	Handle only **GET** requests

📌 Example:

```
app.use(authMiddleware);        // Runs on all routes
app.get('/data', handler);      // Runs only for GET /data
```

28. How do you handle 404 errors in Express?

You define a **catch-all** middleware **after all routes**:

```
app.use((req, res) => {
  res.status(404).json({ message: 'Route not found' });
});
```

29. How do you secure your REST API in Node.js?

- Use **HTTPS**
- Implement **authentication & authorization** (e.g., JWT)
- Sanitize inputs to prevent XSS/SQLi
- Rate-limit requests
- Use helmet for HTTP headers:

```
const helmet = require('helmet');
app.use(helmet());
```

---

30. How do you validate request data in a REST API?

You can use validation libraries to ensure incoming data is valid.

📌 **Example with `express-validator`:**

```
npm install express-validator

const { body, validationResult } = require('express-validator');

app.post('/users',
  body('email').isEmail(),
  (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) return res.status(400).json({ errors:
errors.array() });
    res.send('User created');
  });
```

---

31. What libraries can you use for input validation in Express?

- `express-validator`
- `Joi`
- `yup`
- `zod`

All support schema-based and custom validations.

---

32. How do you handle authentication in a REST API?

Typical options include:

- **JWT tokens**
- **OAuth**
- **API keys**
- **Sessions (less common for APIs)**

📌 Usually, the client includes a token in the `Authorization` header:

```
Authorization: Bearer <token>
```

The server verifies the token on every request.

---

33. What are JWTs and how do you use them?

JWT (JSON Web Token) is a compact, URL-safe token used for securely transmitting information.

- Consists of Header, Payload, and Signature
- Used to verify user identity and permissions

📌 **Generate:**

```
const jwt = require('jsonwebtoken');
const token = jwt.sign({ id: user.id }, 'secret', { expiresIn: '1h'
});
```

📌 **Verify:**

```
jwt.verify(token, 'secret');
```

---

34. How do you implement token-based authentication in Node.js?

1. User logs in → gets JWT token
2. Client sends token with each request
3. Server middleware checks the token

📌 **Middleware example:**

```
function auth(req, res, next) {
  const token = req.headers.authorization?.split(' ')[1];
  if (!token) return res.status(401).send('Unauthorized');

  try {
    const user = jwt.verify(token, 'secret');
    req.user = user;
    next();
  } catch {
    res.status(403).send('Invalid token');
  }
}
```

---

35. How do you handle file uploads in Node.js REST APIs?

Use the `multer` middleware:

```
npm install multer
```

📌 **Example:**

```
const multer = require('multer');
const upload = multer({ dest: 'uploads/' });

app.post('/upload', upload.single('file'), (req, res) => {
  res.send('File uploaded');
});
```

---

36. How do you test REST APIs built with Node.js?

Use testing frameworks:

- `Jest` or `Mocha` for unit tests
- `Supertest` for HTTP API testing

📌 **Example with Supertest:**

```
const request = require('supertest');
request(app)
  .get('/api/users')
  .expect(200)
  .then(res => {
    console.log(res.body);
  });
```

---

37. What is rate limiting and how do you implement it?

Rate limiting prevents abuse by limiting the number of requests a user can make in a given time.

📌 Use `express-rate-limit`:

```
npm install express-rate-limit
```

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

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

app.use(limiter);
```

---

38. How do you log HTTP requests in Node.js?

Use the `morgan` middleware:

```
npm install morgan
```

```
const morgan = require('morgan');
app.use(morgan('dev'));
```

Logs every request with method, status, time, etc.

---

39. What are some best practices for designing REST APIs?

- Use **plural nouns** in endpoints (`/users`, not `/user`)
- Use proper **HTTP methods**
- Send meaningful **status codes**
- Keep URLs simple and consistent
- Use **pagination** for large data
- Protect with **authentication/authorization**
- Use **versioning** (`/api/v1/...`)
- Validate and sanitize all inputs

NPM and module-related Node.js interview questions

1. What is NPM?

NPM (Node Package Manager) is the default package manager for Node.js. It's used to:

- Install and manage packages (libraries or tools)
- Share your own packages
- Handle versioning and dependencies

📦 When you run:

```
npm install express
```

It downloads the **Express.js** package from the NPM registry into your project.

🧠 NPM also:

- Comes pre-installed with Node.js
- Generates and reads a `package.json` file to track dependencies and project metadata

---

2. How do you install a package globally vs locally?

✅ **Local Installation (default):**

Installs the package **in your project folder** under `node_modules`.

```
npm install lodash
```

You can then `require()` it in your code.

🌍 **Global Installation:**

Installs the package **system-wide**, available from the command line anywhere.

```
npm install -g nodemon
```

Global packages are typically CLI tools like `nodemon`, `eslint`, or `typescript`.

📌 To list globally installed packages:

```
npm list -g --depth=0
```

---

3. What is the difference between dependencies and devDependencies?

Type	Used For	Example
`dependencies`	Needed in **production**	express, mongoose
`devDependencies`	Needed **only during development**	nodemon, jest, eslint

You define them in `package.json`:

```
"dependencies": {
  "express": "^4.18.2"
},
"devDependencies": {
  "nodemon": "^3.0.1"
}
```

🧪 Install as a dev dependency:

```
npm install --save-dev nodemon
```

---

4. What is semantic versioning?

Semantic Versioning (semver) is a versioning system for packages using the format:

```
MAJOR.MINOR.PATCH
```

- **MAJOR**: Breaking changes
- **MINOR**: New features, backward-compatible
- **PATCH**: Bug fixes only

📌 Examples:

- `1.0.0`: First stable release
- `1.2.3`: 2 minor updates, 3 patches since version 1.0.0

What do `^` and `~` mean?

- `^1.2.3`: Accept minor and patch updates (e.g., up to `1.9.x`)
- `~1.2.3`: Accept patch updates only (e.g., up to `1.2.x`)

---

5. How do you update a package in Node.js?

🔁 **To update all packages safely (within allowed version range):**

```
npm update
```

⬆️ **To update a specific package to the latest:**

```
npm install express@latest
```

🚀 **Bonus: Automatically update `package.json` with latest versions:**

```
npx npm-check-updates -u
npm install
```

This is super helpful to keep your project up to date.

---

6. How do you create a custom Node.js module?

You can make any `.js` file a reusable module by exporting functions, objects, or classes using `module.exports`.

📦 **Step-by-step example:**

`mathUtils.js` — Custom module

```
function add(a, b) {
  return a + b;
}

function subtract(a, b) {
  return a - b;
}

module.exports = { add, subtract };
```

`app.js` — Use the custom module

```
const math = require('./mathUtils');

console.log(math.add(5, 3));        // Output: 8
console.log(math.subtract(10, 4)); // Output: 6
```

✅ This pattern helps you break code into organized, testable, and reusable pieces — just like using built-in or third-party libraries.

Advanced Node.js Concepts

1. What is clustering in Node.js?

Node.js runs on a **single thread** by default, which means it can handle only one operation at a time per process. **Clustering** allows you to create multiple Node.js processes (workers) that share the same server port. This way, your app can utilize multiple CPU cores and handle more requests concurrently.

📌 Example: Using the built-in `cluster` module, you can fork multiple workers.

---

2. How do you improve performance in a Node.js app?

- Use **clustering** or process managers like PM2 to utilize multiple CPU cores.
- Implement **caching** (in-memory or distributed caches like Redis).
- Use **asynchronous I/O** properly (avoid blocking the event loop).
- Optimize **database queries** and use connection pooling.
- Minimize heavy computations in the main thread (offload with worker threads).
- Use **gzip compression** for responses.
- Properly manage **memory leaks**.
- Use **load balancers** in production.

---

3. What is PM2 and why is it used?

PM2 is a popular production process manager for Node.js applications. It helps you:

- Manage and keep apps alive forever (auto-restart on crashes).

- Run apps in cluster mode easily.
- Monitor resource usage (CPU, memory).
- Handle zero-downtime reloads.

```
npm install pm2 -g
pm2 start app.js -i max  # Runs in cluster mode with max CPU cores
```

---

4. What are worker threads in Node.js?

Worker threads allow you to run JavaScript code in parallel on multiple threads within the same process — useful for CPU-intensive tasks like image processing or complex calculations without blocking the main event loop.

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

const worker = new Worker('./worker.js');
worker.on('message', (msg) => console.log('From worker:', msg));
worker.postMessage('start');
```

---

5. What are child processes and how are they used?

Child processes are separate processes spawned by your Node.js app to run shell commands or other programs, enabling parallel execution.

```
const { exec } = require('child_process');

exec('ls -la', (err, stdout, stderr) => {
  if (err) console.error(err);
  console.log(stdout);
});
```

Useful for running system commands or scripts without blocking your main app.

---

6. What is the role of async/await in Node.js?

`async/await` provides a cleaner syntax for handling asynchronous code, making it look synchronous and easier to read compared to nested callbacks or promise chains.

```
async function fetchData() {
  try {
    const data = await someAsyncFunction();
    console.log(data);
  } catch (err) {
    console.error(err);
  }
}
```

---

7. How does error handling work in async/await?

You handle errors using `try/catch` blocks around `await` statements.

```
async function getUser() {
  try {
    const user = await getUserFromDB();
    return user;
  } catch (error) {
    console.error('Error fetching user:', error);
  }
}
```

Without try/catch, unhandled promise rejections can crash your app.

---

8. What is the difference between synchronous and asynchronous functions?

- **Synchronous** functions block the execution until they finish.

- **Asynchronous** functions allow other code to run while waiting for operations (like I/O) to complete.

```
// Synchronous (blocks event loop)
const data = fs.readFileSync('file.txt');

// Asynchronous (non-blocking)
fs.readFile('file.txt', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

---

9. What is the `util.promisify()` function?

`util.promisify()` converts traditional Node.js callback-style functions into functions that return promises — letting you use `async/await` with them.

```
const util = require('util');
const fs = require('fs');

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

async function read() {
  const content = await readFile('file.txt', 'utf8');
  console.log(content);
}
```

---

10. What is the difference between `spawn()` and `exec()`?

Both create child processes, but:

- `spawn()` streams data (good for large outputs)
- `exec()` buffers data (good for small outputs)

```
const { spawn, exec } = require('child_process');

// spawn example
const ls = spawn('ls', ['-lh', '/usr']);
ls.stdout.on('data', (data) => console.log(`Output: ${data}`));

// exec example
exec('ls -lh /usr', (error, stdout) => console.log(stdout));
```

---

11. What is the EventEmitter class in Node.js?

`EventEmitter` is a core class that allows objects to emit named events and register listeners to respond.

---

12. How do you emit and listen for events?

```
const EventEmitter = require('events');
const emitter = new EventEmitter();

emitter.on('greet', (name) => {
  console.log(`Hello, ${name}!`);
});

emitter.emit('greet', 'Alice');  // Output: Hello, Alice!
```

---

13. What is middleware chaining in Express?

Express allows multiple middleware functions to run sequentially for a request. Each calls `next()` to pass control to the next middleware.

```
app.use((req, res, next) => {
  console.log('Middleware 1');
  next();
});
```

```
app.use((req, res, next) => {
  console.log('Middleware 2');
  res.send('Done');
});
```

---

14. How does Node.js handle concurrency?

Node.js uses a **single-threaded event loop** to handle many I/O operations asynchronously, allowing it to handle thousands of concurrent connections efficiently without creating threads per connection.

---

15. What is the cluster module?

The `cluster` module lets you fork your Node.js process to create worker processes that share the same server port, improving CPU utilization.

Testing Node.js Applications

1. How do you write unit tests in Node.js?

Unit tests check small, isolated pieces of your code (like functions) to make sure they work as expected.

Basic example using Mocha and Chai:

```
// calculator.js
function add(a, b) {
  return a + b;
}
module.exports = add;

// test/calculator.test.js
const add = require('../calculator');
const { expect } = require('chai');
```

```
describe('add function', () => {
  it('should return the sum of two numbers', () => {
    const result = add(2, 3);
    expect(result).to.equal(5);
  });
});
```

Run tests with:

```
mocha
```

This test suite checks if `add(2,3)` equals 5 — simple and effective.

---

2. What are some popular testing frameworks for Node.js?

- **Mocha:** Flexible test runner, widely used.
- **Jest:** Full-featured, zero-config, includes mocks and coverage.
- **Jasmine:** Behavior-driven development framework.
- **AVA:** Minimalistic and fast.
- **Tape:** Simple and small footprint.

---

3. What is Mocha?

Mocha is a **test runner** that executes your test files, organizes tests in suites (`describe`), and allows async testing.

It **does not provide assertions**, so it's often paired with assertion libraries like Chai.

---

4. What is Chai?

Chai is an **assertion library** for Node.js that lets you write readable tests with different styles:

Expect style (most popular):

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

-

Should style:

```
result.should.equal(5);
```

-

Assert style:

```
assert.equal(result, 5);
```

-

---

5. What is Supertest and how is it used?

Supertest is a library for testing HTTP APIs, especially Express apps.

It allows you to simulate HTTP requests and assert on responses.

Example:

```
const request = require('supertest');
const app = require('../app'); // Your Express app

describe('GET /users', () => {
  it('should return 200 and a list of users', (done) => {
    request(app)
      .get('/users')
      .expect(200)
      .expect('Content-Type', /json/)
      .end((err, res) => {
        if (err) return done(err);
        done();
    });
  });
});
```

6. How do you mock dependencies in Node.js tests?

Mocking replaces real dependencies with fake versions to isolate the unit you're testing.

Common tools:

- **Sinon:** For mocks, spies, and stubs.
- **Proxyquire:** Replace dependencies when requiring modules.
- **Jest:** Has built-in mocking capabilities.

Example with Sinon:

```
const sinon = require('sinon');
const myModule = require('../myModule');
const dependency = require('../dependency');

describe('test with mock', () => {
  it('should call dependency once', () => {
    const stub = sinon.stub(dependency, 'someMethod').returns(42);

    const result = myModule.doSomething();
    sinon.assert.calledOnce(stub);

    stub.restore();
  });
});
```

7. What is test coverage and how do you measure it?

Test coverage shows how much of your code is tested (lines, branches, functions).

Tools to measure:

- **Istanbul/nyc:** Most popular coverage tool.
- **Jest:** Has built-in coverage reports.

Run coverage with nyc:

```
nyc mocha
```

It outputs stats like:

- % of lines covered
- % of functions covered
- % of branches covered

Security in Node.js

1. What are common security issues in Node.js applications?

- **Injection attacks** (SQL, NoSQL)
- **Cross-Site Scripting (XSS)**
- **Cross-Site Request Forgery (CSRF)**
- **Broken authentication and session management**
- **Insecure handling of sensitive data (passwords, API keys)**
- **Unvalidated input data**

- **Exposed stack traces or error messages**
- **Security misconfigurations**

---

2. How do you prevent SQL Injection?

- Always use **parameterized queries** or **prepared statements** instead of string concatenation.

Example with MySQL:

```
connection.query('SELECT * FROM users WHERE id = ?', [userId],
callback);
```

- Use ORM libraries like **Sequelize** which handle this automatically.
- Validate and sanitize inputs.

---

3. How do you prevent NoSQL injection?

- Avoid directly inserting user input into queries.
- Use safe query methods (e.g., Mongoose's query APIs).
- Validate and sanitize input.
- For example, don't allow user input to modify query operators like `$gt`, `$ne`.

---

4. What is Helmet and how does it help secure an app?

Helmet is a middleware that sets HTTP headers to protect against common attacks:

- Adds Content Security Policy (CSP)
- Prevents MIME sniffing
- Protects against clickjacking
- Enables HSTS (HTTPS enforcement)

Usage:

```
const helmet = require('helmet');
app.use(helmet());
```

---

5. How do you manage API keys securely?

- Never hardcode keys in your source code.
- Store them in **environment variables** or secure vaults.
- Use `.env` files with `.gitignore` to avoid committing secrets.
- Rotate keys regularly.
- Use scopes/permissions to limit key usage.

---

6. How do you use environment variables in Node.js?

You can access environment variables using `process.env`.

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

These variables are set outside your app, often in your shell or deployment environment.

---

7. What is dotenv and how is it used?

`dotenv` is a library to load environment variables from a `.env` file into `process.env`.

1. Install it:

```
npm install dotenv
```

2. Create a `.env` file:

```
DB_PASSWORD=supersecret
API_KEY=abcdef
```

3. Load it in your app:

```
require('dotenv').config();
console.log(process.env.DB_PASSWORD);
```

---

8. How do you prevent Cross-Site Scripting (XSS)?

- Sanitize user inputs and outputs.
- Use libraries like **DOMPurify** for front-end.
- Use HTTP headers like Content Security Policy (CSP) via Helmet.
- Escape data before rendering in HTML.

---

9. What is CSRF and how do you protect against it?

CSRF (Cross-Site Request Forgery) tricks a user into submitting unwanted requests to a trusted site.

Protection:

- Use CSRF tokens with forms (e.g., `csurf` middleware in Express).
- Implement same-site cookies.
- Require authentication on sensitive endpoints.

Example:

```
const csurf = require('csurf');
app.use(csurf());
```

---

10. How do you secure user passwords?

- Always **hash** passwords before storing (never store plaintext).
- Use strong, slow hashing algorithms like **bcrypt**, **argon2**, or **scrypt**.
- Add a **salt** to each password (bcrypt does this automatically).
- Use libraries like `bcrypt`:

```
const bcrypt = require('bcrypt');
const hash = await bcrypt.hash(password, 10);
```

- When verifying:

```
const match = await bcrypt.compare(inputPassword, storedHash);
```

Database Integration

1. How do you connect Node.js to MongoDB?

You typically use the **MongoDB Node.js driver** or an ODM like Mongoose.

Basic connection example with native driver:

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

async function connect() {
  const uri = 'mongodb://localhost:27017/mydatabase';
  const client = new MongoClient(uri);

  try {
    await client.connect();
    console.log('Connected to MongoDB');
    const db = client.db('mydatabase');
    // Use `db` to query collections
  } catch (err) {
    console.error(err);
  } finally {
    await client.close();
  }
}

connect();
```

---

2. What is Mongoose and how is it used?

Mongoose is an ODM (Object Data Modeling) library for MongoDB in Node.js. It provides:

- Schema-based data modeling
- Validation
- Middleware (hooks)
- Easy querying and relationship management

You define schemas and models to interact with MongoDB more intuitively.

---

3. How do you define schemas and models with Mongoose?

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

const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  age: Number,
  createdAt: { type: Date, default: Date.now }
});

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

// Usage example
async function createUser() {
  const user = new User({ name: 'Alice', email: 'alice@example.com',
age: 25 });
  await user.save();
  console.log('User saved');
}
```

4. How do you connect Node.js with MySQL/PostgreSQL?

You can use native drivers or ORMs like **Sequelize**.

Example with MySQL native driver:

```
const mysql = require('mysql2/promise');

async function connect() {
  const connection = await mysql.createConnection({ host:
'localhost', user: 'root', database: 'test' });
  const [rows] = await connection.execute('SELECT * FROM users');
  console.log(rows);
}
```

5. What is Sequelize?

Sequelize is a popular **ORM** for relational databases like MySQL, PostgreSQL, SQLite, and MSSQL.

It allows you to:

- Define models with JavaScript classes
- Handle migrations and schema changes
- Write complex queries using JavaScript instead of raw SQL

---

6. How do you handle database errors in Node.js?

- Use `try/catch` blocks with async/await to catch exceptions.
- Handle errors in callbacks or promise `.catch()` when using promise-based APIs.
- Log errors for debugging.
- Return meaningful error messages to the client without exposing sensitive info.

Example:

```
try {
  const user = await User.findById(id);
  if (!user) throw new Error('User not found');
} catch (error) {
  console.error(error);
  res.status(500).send('Something went wrong');
}
```

---

7. What are transactions and how are they handled?

Transactions allow multiple database operations to execute atomically — either all succeed or none do — ensuring data consistency.

- In MongoDB (using Mongoose):

```
const session = await mongoose.startSession();
session.startTransaction();

try {
  await User.create([{ name: 'Bob' }], { session });
  await Order.create([{ userId: user._id }], { session });

  await session.commitTransaction();
} catch (error) {
  await session.abortTransaction();
} finally {
  session.endSession();
}
```

- In Sequelize (MySQL/PostgreSQL):

```
const t = await sequelize.transaction();

try {
  await User.create({ name: 'Bob' }, { transaction: t });
  await Order.create({ userId: user.id }, { transaction: t });

  await t.commit();
} catch (error) {
  await t.rollback();
}
```

Deployment & Production

1. How do you deploy a Node.js app to production?

Deploying means getting your app from your local machine to a live server.

Basic steps:

- **Choose a hosting platform**: VPS (DigitalOcean, AWS EC2), PaaS (Heroku, Vercel), or container platforms (Kubernetes, Docker).
- **Set environment variables** securely (don't hardcode secrets).
- **Install dependencies** using `npm install --production`.
- **Run your app with a process manager** (like PM2) for stability.
- **Set up reverse proxy** (using Nginx or Apache) to handle HTTPS, load balancing, and static files.
- **Automate deployments** with scripts or CI/CD pipelines.

---

2. What is process management and why is PM2 useful?

Process management means keeping your app running reliably, restarting it if it crashes, and managing multiple instances.

PM2 is a popular Node.js process manager that:

- Restarts apps on crashes or code changes
- Manages clustering (multi-core usage)
- Offers logs and monitoring dashboards
- Supports zero-downtime reloads

Run your app with PM2 like this:

```
pm2 start app.js
pm2 monit
```

---

3. How do you handle environment variables in production?

- Use real environment variables on the server or container.
- Avoid committing `.env` files to source control.
- In cloud providers, set environment variables in the dashboard.
- Use tools like **dotenv** only in development.
- For sensitive secrets, consider secret managers like AWS Secrets Manager or HashiCorp Vault.

---

4. What is load balancing and how does it apply to Node.js?

Load balancing distributes incoming requests across multiple server instances to:

- Improve performance
- Increase availability and fault tolerance

In Node.js, you can:

- Use the **cluster module** to spawn workers on multiple CPU cores.
- Use external load balancers like **Nginx**, **HAProxy**, or cloud load balancers.
- PM2 also supports clustering with:

```
pm2 start app.js -i max
```

---

5. How do you use Docker with Node.js?

Docker packages your app and its environment into a container for consistent deployment.

Basic steps:

- Create a `Dockerfile`:

```
FROM node:18
WORKDIR /app
COPY package*.json ./
RUN npm install --production
COPY . .
CMD ["node", "app.js"]
```

- Build and run the container:

```
docker build -t my-node-app .
docker run -p 3000:3000 my-node-app
```

---

6. How do you handle logging in production?

- Use logging libraries like **winston** or **pino** for structured logs.
- Separate logs into levels (info, warn, error).
- Output logs to files or external services (Logstash, Datadog, Splunk).
- Implement log rotation to prevent disk space issues.
- Make logs easy to search and analyze.

---

7. How do you monitor performance in a Node.js app?

- Use monitoring tools like **New Relic**, **Datadog**, **AppDynamics**, or open-source tools like **Prometheus + Grafana**.
- Track metrics: response time, CPU/memory usage, error rates.
- Use Node.js built-in profilers or **clinic.js** to analyze performance.
- Set up alerts for anomalies.

---

8. What is the role of nodemon?

`nodemon` is a development tool that automatically restarts your Node.js app when file changes are detected—great for faster development cycles.

It's **not recommended for production**.

Usage:

```
nodemon app.js
```

---

9. What is CI/CD and how can it be applied to a Node.js project?

CI/CD stands for Continuous Integration and Continuous Deployment.

- **CI**: Automatically build and test your Node.js app every time you push code (e.g., GitHub Actions, Jenkins).
- **CD**: Automatically deploy the app to production or staging after tests pass.

Benefits:

- Catch bugs early
- Fast, repeatable releases
- Automated testing and deployment pipelines

Typical pipeline steps:

1. Pull code
2. Install dependencies
3. Run tests
4. Build (optional)

5. Deploy

Tricky & Conceptual Questions

1. What happens when an uncaught exception occurs in Node.js?

If an error (exception) is thrown but **not caught**, Node.js will:

- Print the error stack trace to the console.
- Immediately **terminate the process** to avoid unpredictable behavior.

Why? Because an uncaught exception might leave the app in an inconsistent state.

Best practice: Use `try/catch` for synchronous code, and listen to `'uncaughtException'` or `'unhandledRejection'` events to log errors and gracefully shut down:

```
process.on('uncaughtException', (err) => {
  console.error('Uncaught Exception:', err);
  process.exit(1); // Exit to avoid unstable state
});
```

---

2. Why is Node.js single-threaded?

Node.js uses a **single-threaded event loop** to handle concurrency. This design:

- Simplifies programming by avoiding thread-related bugs like race conditions.
- Uses **non-blocking I/O** so a single thread can handle many connections efficiently.
- Offloads heavy tasks (file I/O, network requests) to background threads in the libuv thread pool.

So, Node.js can handle many tasks concurrently without spawning multiple OS threads for each.

---

3. When should you not use Node.js?

Node.js is awesome for I/O-heavy, real-time apps, but it's not ideal for:

- **CPU-intensive tasks:** Heavy computations block the event loop and slow down all requests.
- **Applications requiring multithreaded parallelism:** Though worker threads exist, Node.js is not designed for parallel CPU-heavy workloads by default.
- **When you need mature libraries for complex domains:** Some domains (like machine learning) have better ecosystems in other languages.

---

4. How does garbage collection work in Node.js?

Node.js uses the **V8 JavaScript engine's garbage collector**, which:

- Automatically frees memory that's no longer referenced.
- Uses a generational GC: young generation (short-lived objects) and old generation (long-lived objects).
- Runs periodically to clean up unused objects.
- Developers usually don't control it directly, but can monitor memory and tune via flags if needed.

---

5. What is the difference between PUT and PATCH in REST APIs?

- **PUT**: Replaces the entire resource with the data sent.
    - If a field is missing in the request, it may get erased.
    - Idempotent (same request repeated yields same result).

- **PATCH**: Applies partial updates to the resource.
  - Only changes the fields specified.
  - Not necessarily idempotent.

Example: Updating user email.

- PUT `/users/1` with `{ "name": "Alice" }` replaces whole user — email might get removed.
- PATCH `/users/1` with `{ "email": "new@example.com" }` updates just the email.

---

6. What is backpressure in Streams?

Backpressure happens when data is produced faster than it can be consumed downstream.

In Node.js streams, it's a built-in mechanism to:

- **Pause the readable stream** when the writable stream is overwhelmed.
- Prevent memory overload and crashes.

This flow control allows producers and consumers to work in sync.

---

7. How does the async queue work in Node.js?

Node.js uses the **event loop** with phases (timers, I/O callbacks, idle, poll, check, close callbacks) to manage async tasks.

Async queues (e.g., with libraries like `async.queue`) allow you to:

- Control concurrency (limit how many async tasks run simultaneously).
- Queue tasks and process them in order.

Example with async library:

```
const async = require('async');

const queue = async.queue(async (task) => {
  await doWork(task);
}, 2); // concurrency = 2

queue.push({ id: 1 });
queue.push({ id: 2 });
```

Node.js internally uses **libuv's thread pool** for async I/O, but the event loop manages task scheduling on the single main thread.

Additional Important Node.js Questions & Answers

Core Concepts & Runtime

1. What are `process.nextTick()` and `setImmediate()`? How do they differ?

- `process.nextTick()` queues a callback to be invoked **immediately after the current operation completes**, before the event loop continues.
- `setImmediate()` queues a callback to run on the **next iteration** of the event loop.

Example:

```
process.nextTick(() => console.log('nextTick'));
setImmediate(() => console.log('setImmediate'));
console.log('sync');
```

Output:

```
sync
nextTick
setImmediate
```

`nextTick` runs before any I/O or timers; `setImmediate` runs after I/O callbacks.

---

2. What is the `EventEmitter` class? How do you create and use custom events?

`EventEmitter` allows objects to emit named events and listen for them.

Example:

```
const EventEmitter = require('events');
const emitter = new EventEmitter();

emitter.on('greet', (name) => {
  console.log(`Hello, ${name}!`);
});

emitter.emit('greet', 'Alice');
```

Output: `Hello, Alice!`

It's fundamental for asynchronous communication in Node.js.

---

3. How does Node.js handle asynchronous code internally? Explain the event loop phases.

Node.js uses the **event loop** to handle async tasks without blocking.

Main phases:

- **Timers:** Executes callbacks scheduled by `setTimeout` and `setInterval`.
- **Pending callbacks:** Executes I/O callbacks deferred to next iteration.

- **Idle, prepare:** Internal operations.
- **Poll:** Retrieves new I/O events; executes I/O callbacks.
- **Check:** Executes callbacks scheduled by `setImmediate`.
- **Close callbacks:** Handles closed connections.

Tasks are processed in this order each loop iteration.

---

4. What are worker threads in Node.js and when should you use them?

Worker threads run JavaScript code in **parallel threads** — useful for CPU-intensive tasks that block the event loop.

Use them when your app needs heavy computation without blocking other requests.

---

5. Explain the difference between callbacks, promises, and async/await.

- **Callbacks:** Functions passed as arguments, executed when async operation finishes. Can lead to “callback hell.”
- **Promises:** Objects representing future results; allow chaining with `.then()`.
- **Async/await:** Syntactic sugar over promises; lets you write async code that looks synchronous, improving readability.

---

6. How does Node.js handle memory management and what tools can you use to detect memory leaks?

V8 engine manages memory automatically with garbage collection.

To detect leaks:

- Use tools like **Chrome DevTools**, **node --inspect**, or **heapdump**.

- Monitor memory usage over time.
- Look for increasing memory without release, which signals leaks.

Modules & Package Management

7. What is the difference between CommonJS and ES Modules (ESM) in Node.js?

- **CommonJS (CJS):** Uses `require()` and `module.exports`. Synchronous module loading.
- **ES Modules:** Uses `import`/`export` syntax. Supports asynchronous and static analysis.

ESM is the modern standard but Node.js supports both.

---

8. How does Node.js resolve modules when you call `require()`?

Node.js searches in this order:

1. Core modules (like `fs`).
2. File or directory modules relative to current file.
3. Inside `node_modules` folders up the directory chain.
4. If not found, throws a module not found error.

---

9. What is the purpose of the `main` field in package.json?

It specifies the **entry point** file of a package (default is `index.js`).

When someone imports your package, Node.js loads the file in `main`.

---

10. How do you publish a Node.js package to npm?

Steps:

1. Create an npm account.
2. Add `package.json` and code.
3. Run `npm login` to authenticate.
4. Run `npm publish` to upload your package.

Performance & Scalability

11. How do you optimize Node.js applications for high throughput?

- Avoid blocking the event loop.
- Use asynchronous APIs.
- Use clustering or worker threads.
- Cache results where possible.
- Profile and fix bottlenecks.
- Use load balancers for scaling horizontally.

---

12. What is event loop blocking, and how do you detect it?

Blocking happens when synchronous code takes too long, preventing other events from processing.

Detect with tools like:

- `clinic.js` — Event Loop Delay tool.

- Manual timing (`setInterval` to check delay).

---

13. Explain clustering and how it improves Node.js app performance.

Clustering runs multiple Node.js instances on different CPU cores sharing the same server port.

This spreads load and uses full CPU capacity, increasing concurrency.

---

14. How do you scale Node.js horizontally?

- Deploy multiple instances on different machines or containers.
- Use a load balancer to distribute traffic.
- Share state externally (e.g., Redis) since instances are stateless.

---

15. What are some common causes of memory leaks in Node.js?

- Global variables holding references.
- Event listeners not removed.
- Closures holding onto variables.
- Caches growing without limits.

Security & Best Practices

16. How do you prevent Denial of Service (DoS) attacks in Node.js?

- Use rate limiting.

- Validate and sanitize inputs.
- Avoid blocking event loop.
- Use security middleware like **helmet**.
- Use a reverse proxy with DDoS protection.

---

17. What is input validation and why is it important?

Ensuring incoming data matches expected format to avoid injection attacks, crashes, or invalid data storage.

---

18. How do you sanitize user input to avoid security risks?

- Strip out dangerous characters.
- Use libraries like **validator.js**.
- Escape output in HTML contexts.

---

19. Explain CORS and how to configure it in Node.js.

Cross-Origin Resource Sharing controls which domains can access your API.

Configure with the **cors** package:

```
const cors = require('cors');
app.use(cors({ origin: 'https://example.com' }));
```

---

20. How do you securely handle file uploads in Node.js?

- Validate file types and sizes.

- Store files outside of the web root.
- Use libraries like **multer**.
- Scan files for malware.

REST API & Web Development

21. How do you implement pagination in REST APIs?

Use query parameters like `?page=2&limit=10` to return chunks of data instead of all at once.

---

22. What is idempotency and why is it important in REST APIs?

An operation is idempotent if repeating it has the same effect as doing it once (e.g., PUT).

Important for safe retries.

---

23. How do you implement rate limiting in a Node.js API?

Use middleware like **express-rate-limit** to limit the number of requests per IP.

---

24. What are websockets and how do they work with Node.js?

Websockets enable two-way real-time communication over a single TCP connection.

Use libraries like **socket.io**.

---

25. What is the difference between stateless and stateful applications?

- **Stateless:** Each request is independent; no session stored on server.
- **Stateful:** Server keeps session info (like login status).

Stateless apps scale easier.

Testing & Debugging

26. How do you debug a Node.js application?

- Use `console.log` for quick checks.
- Use `node --inspect` and Chrome DevTools.
- Use debuggers in IDEs (VSCode).
- Use profiling tools like **clinic.js**.

---

27. What is Test-Driven Development (TDD) and how do you apply it to Node.js?

Write tests **before** code, then develop just enough to pass tests. Use frameworks like Mocha or Jest.

---

28. How do you mock HTTP requests in tests?

Use libraries like **nock** to intercept and mock HTTP calls.

---

29. How can you perform load testing on your Node.js API?

Use tools like **Apache JMeter**, **k6**, or **Artillery** to simulate many users and measure performance.

Ecosystem & Tools

30. What is nvm and why is it useful?

Node Version Manager lets you install and switch between Node.js versions easily.

---

31. How does Node.js versioning work?

Uses semantic versioning: `MAJOR.MINOR.PATCH`. New major versions can include breaking changes.

---

32. What is the difference between npm, yarn, and pnpm?

- **npm:** Default Node package manager.
- **yarn:** Faster installs, better caching.
- **pnpm:** Efficient disk usage by linking packages.

---

33. What is the role of Babel or TypeScript in Node.js projects?

- **Babel:** Transpiles modern JS to compatible versions.
- **TypeScript:** Adds static typing and compiles to JS.