Node.js

1. What is Node.js?

Node.js is a **JavaScript runtime environment** that allows you to run JavaScript code **outside of a web browser**—typically on a server. It's built on Chrome's V8 JavaScript engine, which makes it fast and efficient.

Key points about Node.js:

- **Server-side JavaScript:** Traditionally, JavaScript was only used in browsers for client-side scripting. Node.js lets you use JavaScript for backend development too.
- **Event-driven and non-blocking:** Node.js uses an event-driven, asynchronous programming model, which means it can handle many operations at once without waiting for any single task to finish (great for scalable network apps).
- **Built-in modules:** It provides many modules to handle things like file systems, HTTP servers, streams, etc.
- **NPM (Node Package Manager):** Comes with npm, a huge ecosystem of open-source libraries and tools you can easily add to your projects.
- **Use cases:** Web servers, APIs, real-time chat applications, tools, automation scripts, and much more.

2. Installing Node.js

Download from nodejs.org and install. After installation, verify:

```
node -v

npm -v
```

3. Example

Create a file app.js:

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

4. Node.js Modules

- A module is a reusable block of code whose scope is private by default.
- Node.js treats each file as a separate module.
- Modules help organize code into smaller, manageable pieces.
- Modules can export functions, objects, or variables that other files can import and use.

Types of Modules in Node.js

1. **Core Modules** — Built-in modules shipped with Node.js (e.g., fs, http, path).
2. **Local/Custom Modules** — Modules created by you in your project files.
3. **Third-party Modules** — Modules published by others and installed via npm (e.g., express, lodash).

5. Event Loop & Asynchronous Programming

Node.js is built on Chrome's V8 JavaScript engine and uses an event-driven, non-blocking I/O model. The event loop is central to this model.

Key Points about Node.js Event Loop

- Node.js runs JavaScript on a single thread.
- It uses the libuv library to handle asynchronous operations like file I/O, networking, timers, etc.
- The event loop handles multiple phases, managing callbacks and executing them efficiently.

6. Callbacks

A callback is a function passed as an argument to another function, which is then invoked inside the outer function to complete some kind of routine or action.

Why Callbacks in Node.js?

Node.js is asynchronous and non-blocking. Instead of waiting for a task (like reading a file or querying a database) to finish before moving on, Node.js uses callbacks to handle these tasks. This way, the program can continue running while waiting for operations to complete.

Example:

```
function greet(name, callback) {
  console.log('Hello ' + name);
  callback();
}
function sayGoodbye() {
  console.log('Goodbye!');
}
greet('Alice', sayGoodbye);
```

7. Promises

A **Promise** is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

Instead of using callbacks, Promises provide a cleaner, more manageable way to handle async operations.

Promise States

- **Pending**: Initial state, neither fulfilled nor rejected.
- **Fulfilled**: The operation completed successfully.
- **Rejected**: The operation failed.

Creating and using:

```
const myPromise = new Promise((resolve, reject) => {
  const success = true;  // simulate success or failure
  if (success) {
    resolve('Operation succeeded!');
  } else {
    reject('Operation failed.');
  }
});
myPromise
  .then(result => {
    console.log(result);  // runs if promise is fulfilled
  })
  .catch(error => {
    console.error(error); // runs if promise is rejected
  });
```

8. Async/Await

async/await is syntactic sugar built on top of Promises that makes asynchronous code look and behave more like synchronous code — easier to read and write.

- async marks a function as asynchronous.
- await pauses the execution of the async function until the Promise is resolved or rejected.

Example:

```
function waitOneSecond() {
  return new Promise(resolve => {
  setTimeout(() => {
      resolve('Done waiting!');
    }, 1000);
  });
}
async function myAsyncFunction() {
  console.log('Start');
  const result = await waitOneSecond();
  console.log(result);  // Prints "Done waiting!" after 1 second
  console.log('End');
}
myAsyncFunction();
```

Why use async/await?

- Makes asynchronous code easier to understand and maintain.
- Avoids chaining .then() calls.
- Can handle errors with simple try/catch blocks.

9. File System (fs) Module

The fs module in Node.js provides an API to interact with the file system. It allows you to:

- Read files
- Write files
- Update files

- Delete files
- Work with directories
- And much more

10. HTTP Module - Creating a Server

In Node.js, the **http module** lets you create a web server that can handle requests and send responses. Here's the step-by-step explanation and example.

1. Import the http module

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

2. Create the server

```
const server = http.createServer((req, res) => {
  // Set response header
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  // Send response body
  res.end('Hello, Node.js Server!');
});
```

3. Listen on a port

```
server.listen(3000, () => {
  console.log('Server is running at http://localhost:3000');
});
```

11. Express.js Framework

- A **minimal and flexible** Node.js framework for building web applications and APIs.
- It simplifies HTTP server creation (compared to the raw http module).
- Provides built-in features like:
    - **Routing** (handling different URLs easily)
    - **Middleware** (functions to process requests before sending a response)
    - **Error handling**
    - Integration with **templates** and **databases**

12. Middleware in Express

A middleware function in Express has the signature:

```
(req, res, next) => { ... }
```

- **req** → Request object
- **res** → Response object
- **next** → Function that passes control to the next middleware in the stack

If you **don’t call next()** (and don’t send a response), the request will be stuck.

Types of Middleware

1. **Application-level** middleware → Used throughout the app
2. **Router-level** middleware → Applied only to specific routes
3. **Built-in** middleware → e.g., express.json(), express.static()
4. **Third-party** middleware → e.g., morgan, cors

13. Working with JSON Data

1. Enable JSON Parsing Middleware

Before you can read JSON from incoming requests, you need:

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

2. Sending JSON Data (Response)

You can send JSON with:

```
res.json({ message: 'Hello, JSON!' });
```

3. Receiving JSON Data (Request)

```
const express = require('express');
const app = express();
// Middleware to parse JSON
app.use(express.json());
// POST route to receive JSON
app.post('/data', (req, res) => {
  console.log(req.body); // Logs the JSON data sent by the client
```

```
    res.json({
        status: 'success',
        received: req.body
    });
});
// Start server
app.listen(3000, () => console.log('Server running on port 3000'));
```

14. Environment Variables

Environment variables are values stored outside your code that you can use to configure your app — without hardcoding secrets or settings.

1. Why Use Environment Variables?

- Keep **sensitive data** (API keys, passwords) out of your code
- Change settings between **development**, **testing**, and **production**
- Avoid editing source files for different environment

15. Node Package Manager (npm)

Node Package Manager (NPM) is the default package manager for **Node.js** — it's how you install, manage, and share JavaScript code libraries.

1. What is NPM?

- **Comes with Node.js** (no separate install needed)
- Lets you install packages (dependencies) from the **npm registry**
- Can also publish your own packages
- Manages dependencies via package.json

16. Debugging Node.js

Debugging in **Node.js** means finding and fixing errors (bugs) in your JavaScript code while it's running on the server.
There are several ways to do it — from simple console.log() checks to advanced debugging tools.

Example:

```
const add = (a, b) => {

  console.log('a:', a, 'b:', b); // Debug print

  return a + b;

};

console.log(add(5, 3));
```

17. Streams

are a way to **read or write data piece-by-piece** (chunks) instead of loading it all into memory at once.
They're especially useful for large files, network data, or continuous data flows.

Why Streams?

Without streams:

- Read file → entire content in memory → process it
- Bad for very large files (can crash your app due to memory usage)

With streams:

- Read file **chunk-by-chunk**
- Process each chunk immediately
- Much more **efficient and scalable**

Types of Streams

1. **Readable** → for reading data (e.g., fs.createReadStream)
2. **Writable** → for writing data (e.g., fs.createWriteStream)
3. **Duplex** → can read & write (e.g., network sockets)
4. **Transform** → like duplex, but can modify data as it passes through (e.g., zlib compression)

Example:

```
const fs = require('fs');
// Create a readable stream (source file)
const readableStream = fs.createReadStream('input.txt', 'utf8');
// Create a writable stream (destination file)
const writableStream = fs.createWriteStream('output.txt');
// Pipe data from readable stream to writable stream
readableStream.pipe(writableStream);
console.log('File copy started...');
readableStream.on('end', () => {
  console.log('File copy completed.');
});
```

18. Buffer

- A Buffer is a temporary storage space for binary data, used when dealing with streams, files, or network data.
- **Why it exists:**
  JavaScript (and V8) handles only **Unicode strings**, but streams deal with **raw binary data**. The Buffer object bridges this gap.
- **Where it's used:**
  - Reading from or writing to files
  - Working with TCP/HTTP streams
  - Dealing with binary formats like images, videos, PDFs

Creating Buffer:

```
// From a string
const buf1 = Buffer.from('Hello', 'utf8');
console.log(buf1);           // <Buffer 48 65 6c 6c 6f>
console.log(buf1.toString()); // Hello
// Allocating a buffer of size 10 bytes (filled with zeros)
const buf2 = Buffer.alloc(10);
console.log(buf2);
// From an array of bytes
```

```
const buf3 = Buffer.from([72, 101, 108, 108, 111]);
console.log(buf3.toString()); // Hello
```

19. Process Object

- process is a **global object** in Node.js that provides information about, and control over, the **current Node.js process**.
- **Availability:**
  It's always available — no need to require() it.
- **Main uses:**
  - Access **command-line arguments**
  - Work with **environment variables**
  - Handle **events** (like exit, signals)
  - Control the **process lifecycle** (exit, kill)
  - Get system-related info (memory, CPU usage)

Example:

```
// process_info.js
console.log('Process ID:', process.pid);
console.log('Node.js Version:', process.version);
console.log('Platform:', process.platform);
console.log('Current Working Directory:', process.cwd());
console.log('Uptime (seconds):', process.uptime());
console.log('Command-line Arguments:', process.argv);
console.log('Environment Variables (sample):', process.env.USER || process.env.USERNAME);
console.log('Memory Usage:', process.memoryUsage());
// Run a callback before event loop continues
process.nextTick(() => {
  console.log('This runs before the event loop continues.');
});
// Exit the process after 2 seconds
setTimeout(() => {
console.log('Exiting process...');
```

```
  process.exit(0); // 0 means success, non-zero means error
}, 2000);
```

20. Child Processes

Child processes allow you to run other programs or scripts from your Node.js application.

- They are useful for performing CPU-intensive tasks, executing system commands, or running multiple processes simultaneously.
- Node.js provides the **child_process** module to create and manage them.

Main Methods of child_process

1. exec() – Run a command in a shell

- Best for running short commands and getting the complete output as a buffer.

2. spawn() – Stream data from a command

- Best for **large output** or continuous data (e.g., running a long process).

3. execFile() – Run an executable file directly

4. fork() – Create a new Node.js process

- Used to run another JavaScript file as a separate Node.js process.
- Allows **inter-process communication (IPC)** between parent and child.

21. Working with Databases

1. Choosing a Database Type

Node.js can work with:

- **Relational Databases (SQL)** → MySQL, PostgreSQL, MariaDB, SQLite, etc.
- **NoSQL Databases** → MongoDB, Redis, Cassandra, etc.

2. Installing Required Packages

For MySQL:

```
npm install mysql2
```

3. **Connecting to a Database**

```
const mysql = require('mysql2');
// Create connection
const db = mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: 'yourpassword',
    database: 'testdb'
});
// Connect
db.connect(err => {
    if (err) throw err;
    console.log('Connected to MySQL!');
});
```

22. Error Handling

- Node.js is asynchronous and event-driven — unhandled errors can **crash your app**.
- Good error handling makes your application **more reliable, secure, and easier to debug**.

Types of Errors in Node.js

1. **Operational Errors** → Expected errors that can happen during normal operation.
   - Examples: Database connection failed, file not found, invalid user input.
2. **Programmer Errors** → Bugs in code.
   - Examples: Undefined variables, syntax mistakes, logic errors.

Example:

```
app.use((err, req, res, next) => {
    console.error(err.stack);
    res.status(500).json({ message: 'Something went wrong!' });
});
```

23. Working with NPM Scripts

NPM scripts are **custom commands** you define inside the package.json file to automate tasks like:

- Running your app
- Building your project
- Running tests
- Linting code
- Deploying

They help you avoid long terminal commands by giving them a **short alias**.

Example:

```
{
  "name": "my-app",
  "version": "1.0.0",
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js",
    "test": "echo \"Running tests...\" && exit 0"
  }
}
```

24. Creating a REST API

A **REST API (Representational State Transfer)** is a way to let clients (like web apps, mobile apps, or other servers) interact with your server via HTTP methods:

- **GET** → Read data
- **POST** → Create data
- **PUT/PATCH** → Update data
- **DELETE** → Delete data

Basic REST API Structure

```
rest-api-example/
│
├── index.js        # Entry point
├── routes/         # Route files
└── data/           # Mock or DB files
```

Example:

```
const express = require('express');
const app = express();
const PORT = 3000;
// Middleware to parse JSON
app.use(express.json());
// Sample in-memory data
let users = [
  { id: 1, name: 'John Doe', email: 'john@example.com' },
  { id: 2, name: 'Jane Doe', email: 'jane@example.com' }
];
// GET all users
app.get('/api/users', (req, res) => {
  res.json(users);
});
// GET a single user
app.get('/api/users/:id', (req, res) => {
  const user = users.find(u => u.id === parseInt(req.params.id));
  if (!user) return res.status(404).json({ message: 'User not found' });
  res.json(user);
});
// CREATE a new user
app.post('/api/users', (req, res) => {
  const newUser = {
    id: users.length + 1,
        name: req.body.name,
        email: req.body.email
    };
    users.push(newUser);
    res.status(201).json(newUser);
});
// UPDATE a user
app.put('/api/users/:id', (req, res) => {
    const user = users.find(u => u.id === parseInt(req.params.id));
    if (!user) return res.status(404).json({ message: 'User not found' });
    user.name = req.body.name || user.name;
    user.email = req.body.email || user.email;
    res.json(user);
});
// DELETE a user
app.delete('/api/users/:id', (req, res) => {
    users = users.filter(u => u.id !== parseInt(req.params.id));
    res.json({ message: 'User deleted' });
});
// Start server
app.listen(PORT, () => console.log(`Server running at http://localhost:${PORT}`));
```

25. Security Basics

1. Keep Dependencies Secure

- **Why:** Many attacks target vulnerable NPM packages.
- **How:**
  - Use npm audit to check for vulnerabilities.
  - Prefer well-maintained libraries.
  - Update dependencies regularly (npm update).
  - Avoid installing unnecessary packages.

2. Avoid Exposing Sensitive Data

- **Why:** API keys, database credentials, and secrets can be stolen.
- **How:**
    - Store secrets in **environment variables** (process.env).
    - Use .env files with dotenv package (never commit them to Git).
    - Use secret managers in production (AWS Secrets Manager, Vault).

```
require('dotenv').config();

const dbPassword = process.env.DB_PASSWORD;
```

3. Validate and Sanitize Input

- **Why:** Prevents **SQL Injection**, **XSS**, and **command injection**.
- **How:**
    - Use libraries like validator or joi for input validation.
    - Escape special characters in output.
    - Always validate API request bodies, query params, and headers.

```
const Joi = require('joi');

const schema = Joi.object({

  username: Joi.string().alphanum().min(3).required()

});
```

4. Use HTTPS

- **Why:** Prevents man-in-the-middle (MITM) attacks.
- **How:**
    - Use TLS/SSL certificates (Let’s Encrypt for free).
    - Redirect HTTP to HTTPS in production.

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

5. Protect Against Cross-Site Scripting (XSS)

- **Why:** Prevent attackers from injecting malicious scripts.
- **How:**
    - Use templating engines that auto-escape HTML (e.g., Handlebars, EJS).
    - Sanitize user-generated content (xss npm package).
    - Set HTTP header Content-Security-Policy (CSP).

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

app.use(helmet());
```

6. Prevent NoSQL Injection

- **Why:** MongoDB and other NoSQL DBs can also be attacked.
- **How:**
  - Never directly inject user input into queries.
  - Use parameterized queries and validation.

```
// BAD
db.collection('users').find({ username: req.body.username });
// GOOD
db.collection('users').find({ username: sanitizedUsername });
```

7. Prevent Cross-Site Request Forgery (CSRF)

- **Why:** Stops attackers from making unauthorized requests on behalf of users.
- **How:**
  - Use csurf middleware in Express for POST requests.
  - Implement SameSite cookies.

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

8. Limit Request Rate (Rate Limiting)

- **Why:** Prevents brute force & DDoS attacks.
- **How:**
  - Use express-rate-limit to block excessive requests.

```
const rateLimit = require('express-rate-limit');
app.use(rateLimit({ windowMs: 15 * 60 * 1000, max: 100 }));
```

9. Handle Errors Securely

- **Why:** Error messages can reveal sensitive info.
- **How:**
  - Don’t expose stack traces in production.
  - Send generic error messages to users, log details internally.

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something went wrong!');
});
```

10. Keep Node.js & Packages Updated

- **Why:** Security patches fix known vulnerabilities.
- **How:**
    - Upgrade Node.js regularly.
    - Subscribe to Node.js security alerts.

26. Testing with Mocha and Chai

- **Mocha** → A JavaScript test framework for Node.js (runs the tests).
- **Chai** → An assertion library (helps write readable test conditions like *should*, *expect*, *assert*).

Installing Mocha & Chai

```
npm install --save-dev mocha chai
```

Structure

```
my-app/
│── index.js
│── test/
│   └── index.test.js
```

Example:

```
const { expect } = require('chai');
const { add } = require('../index');
describe('Math functions', () => {
  it('should return sum of two numbers', () => {
    expect(add(2, 3)).to.equal(5);
  });
  it('should return a number', () => {
    expect(add(2, 3)).to.be.a('number');
  });
});
```

27. Advanced Concepts

1. Event Loop & Asynchronous Patterns

- **Event Loop** handles async tasks without blocking the main thread.
- Patterns:
    - **Callbacks** (older)
    - **Promises** (modern)
    - **async/await** (syntactic sugar over Promises)

Example:

```
console.log("Start");
setTimeout(() => console.log("Async Task Done"), 0);
console.log("End");
// Output: Start → End → Async Task Done
```

2. Streams & Buffers

- **Streams**: Efficiently read/write large data in chunks.
- **Buffers**: Raw binary data handling.

Example:

```
const fs = require('fs');
const readStream = fs.createReadStream('bigfile.txt', 'utf8');
readStream.on('data', chunk => console.log('Chunk:', chunk.length));
readStream.on('end', () => console.log('File read complete'));
```

3. Child Processes & Worker Threads

- **Child Processes**: Run shell commands or spawn new Node processes.
- **Worker Threads**: Run JS code in parallel for CPU-intensive tasks.

Example:

```
const { exec } = require('child_process');
exec('ls', (err, stdout) => {
  if (err) throw err;
  console.log(stdout);
});
```

4. Clustering for Scalability

- **Cluster Module**: Run multiple instances of your app to utilize all CPU cores.

Example:

```
const cluster = require('cluster');
const os = require('os');
if (cluster.isMaster) {
  for (let i = 0; i < os.cpus().length; i++) {
    cluster.fork();
  }
} else {
  require('./server'); // Your express app
}
```

4. Clustering for Scalability

- **Cluster Module**: Run multiple instances of your app to utilize all CPU cores.

Example:

```
const cluster = require('cluster');
const os = require('os');
if (cluster.isMaster) {
  for (let i = 0; i < os.cpus().length; i++) {
    cluster.fork();
  }
} else {
  require('./server'); // Your express app
}
```

5. Middleware & Request Lifecycle

- Custom middleware functions in Express handle requests step-by-step.

Example:

```
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next();
});
```

6. Security in Node.js

- Prevent **SQL Injection**, **XSS**, and **CSRF**.
- Use packages:
  - helmet (security headers)
  - express-rate-limit (limit requests)
  - validator (sanitize input)

7. Caching & Performance

- Cache data with **Redis** or **memory cache** to reduce DB load.

Example:

```
const Redis = require('redis');
const client = Redis.createClient();
client.set('key', 'value');
client.get('key', (err, val) => console.log(val));
```

8. Environment Management

- Use .env files and dotenv to store secrets.

Example:

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

9. Advanced Testing

- Use **Mocha + Chai + Supertest** for full API tests.
- Mock dependencies with sinon.

10. Logging & Monitoring

- Use winston or pino for structured logs.
- Integrate with monitoring tools like **PM2**, **New Relic**, or **Datadog**.

11. Deploying Node.js Apps

- **PM2** process manager
- Dockerizing your Node app
- CI/CD pipelines

12. Advanced Error Handling

- Centralized error handler

- Graceful shutdown on fatal errors

Example:

```
process.on('unhandledRejection', (err) => {
  console.error('Unhandled Rejection', err);
  process.exit(1);
});
```

Summary

Node.js is versatile for:

- Web servers
- REST APIs
- Real-time apps
- CLI tools
- MicroservicesIts asynchronous, event-driven nature makes it efficient for scalable applications.