NODE.JS

DEVELOPER

NOTES

👨🏻‍💻 Name: Mr. Rupesh Lal

🌐 Website: www.itmodem.com

📞 WhatsApp: +91 7870886355

💼 Profession: Software Engineer

**🧠 MASTER NOTES OUTLINE – Node.js, Express.js, MongoDB**

**Structure for each topic**:

✅ *Theory (Concepts)*

✅ *Syntax & Code Examples*

✅ *Exercises*

✅ *Mini Projects*

✅ *Interview Questions*

**🔵 SECTION 1: Node.js – Basics to Advanced**

**1. Introduction to Node.js**

* What is Node.js? Why use it?
* Node vs Browser
* Node.js Architecture
* Installing Node.js & npm

**2. Node.js Fundamentals**

* Modules (Built-in, Custom, Third-party)
* CommonJS vs ES Modules
* require, module.exports, import, export
* File system module (fs)
* Path module
* Events module
* OS module
* process object
* Streams and Buffers
* CLI tools

**3. Asynchronous Programming in Node**

* Callback functions
* Promises
* async/await
* Event Loop & Call Stack
* Error handling

**4. HTTP & Server Basics**

* Creating a basic HTTP server
* Handling requests/responses
* Status codes
* Serving HTML/JSON

**5. npm & Package Management**

* package.json & package-lock.json
* Installing packages
* Dev dependencies vs Prod dependencies
* Versioning (^, ~)

**6. Debugging & Logging**

* console.log, debug, node inspect, VSCode debugger
* Logging best practices

**🔸 Exercises: Create a file reader, HTTP server**

**🔸 Mini Project: Simple File Upload Server**

**🔸 Interview Qs: What is event-driven programming in Node?**

**📘Chapter 1. Introduction to Node.js**

**✅ What is Node.js?**

* **Node.js** is an **open-source**, **cross-platform** JavaScript runtime environment.
* It allows you to **run JavaScript code outside the browser**, primarily on the **server-side**.
* Built on **Google Chrome’s V8 JavaScript engine**.
* Created by **Ryan Dahl** in **2009**.

**Think of it like this**: Node.js lets you use JavaScript to write backend (server-side) code just like you use it on the frontend (browser).

**✅ Why Use Node.js?**

| **Feature** | **Description** |
| --- | --- |
| 🔁 Non-blocking I/O | Node uses **asynchronous** event-driven architecture. No waiting for one task to complete. |
| 🚀 Fast Performance | Built on V8 engine which compiles JavaScript to native machine code. |
| 🔗 Single Language | Use **JavaScript for both frontend and backend**, reducing learning curve. |
| 📦 Rich Ecosystem | Access to thousands of open-source libraries via **npm**. |
| ⚙️ Scalable | Good for building **real-time applications** (like chat apps, gaming, API services). |

🧠 Example: Apps like **Netflix, LinkedIn, PayPal** use Node.js for performance and scalability.

**✅ Node.js vs Browser JavaScript**

| **Feature** | **Browser** | **Node.js** |
| --- | --- | --- |
| Environment | Runs in browser | Runs on server |
| Global Object | window | global |
| APIs Available | DOM, alert, document | File system, HTTP, OS, Streams |
| Use Case | UI Interactions | Server, API, File handling |

**✅ Node.js Architecture**

Node.js uses a **Single-threaded Event Loop** architecture.

**Core Components:**

1. **Event Loop** – Manages async tasks using events and callbacks.
2. **V8 Engine** – Executes JavaScript code.
3. **libuv** – Provides async I/O, thread pool, and event loop.
4. **C++ Bindings** – Bridges low-level system access.
5. **Node APIs** – Expose functionalities like fs, http, os.

**Flow Example**: Client sends request → Node handles it using Event Loop → Uses callback/Promise → Sends response (non-blocking)

🖼️ *(Add diagram if needed: Event Loop → Callback Queue → Call Stack → Response)*

**✅ Installing Node.js and npm**

1. **Download Node.js**
   * Visit: [https://nodejs.org](https://nodejs.org/)
   * Choose **LTS version** (recommended for most users)
2. **Install it (Windows/macOS/Linux)**
3. **Verify Installation**

node -v *# shows Node.js version*

npm -v *# shows npm version*

1. **Basic Commands**

node *# enters Node REPL (interactive shell)*

node app.js *# runs JavaScript file*

npm init *# initializes package.json*

npm install <package> *# installs a package*

**📝 Summary**

* Node.js lets you build server-side apps using JavaScript.
* It’s fast, scalable, and non-blocking.
* Ideal for real-time, I/O-heavy applications.
* Installs with npm (Node Package Manager) for managing packages.

**📘Chapter 2. Node.js Fundamentals**

**✅ 1. Modules in Node.js**

Node.js follows a **modular architecture**. Code is split into reusable parts called **modules**.

**a. 🔹 Built-in Modules**

Node provides core modules like:

* fs – File system operations
* path – Work with file and directory paths
* http – Create HTTP server
* events – Event-driven programming
* os – System information

**Example**:

const fs = require('fs');

**b. 🔹 Custom Modules**

Any .js file is a module.

*// greet.js*

function greet(name) {

return `Hello, ${name}`;

}

module.exports = greet;

*// app.js*

const greet = require('./greet');

console.log(greet('John'));

**c. 🔹 Third-party Modules**

Installed via **npm**.

npm install lodash

const \_ = require('lodash');

**✅ 2. CommonJS vs ES Modules**

| **Feature** | **CommonJS (require)** | **ES Modules (import)** |
| --- | --- | --- |
| File Extension | .js | .mjs or set "type": "module" in package.json |
| Import Syntax | const fs = require('fs') | import fs from 'fs' |
| Export Syntax | module.exports = value | export default value or export {} |
| Execution | Synchronous | Asynchronous |

Node uses **CommonJS by default**.

**✅ 3. require, module.exports, import, export**

**🔹 CommonJS:**

*// math.js*

function add(a, b) {

return a + b;

}

module.exports = add;

*// app.js*

const add = require('./math');

console.log(add(2, 3));

**🔹 ES Modules:**

*// math.mjs*

export function add(a, b) {

return a + b;

}

*// app.mjs*

import { add } from './math.mjs';

console.log(add(2, 3));

**✅ 4. File System Module (fs)**

Use fs to read/write files.

**Synchronous:**

const data = fs.readFileSync('file.txt', 'utf8');

**Asynchronous:**

fs.readFile('file.txt', 'utf8', (err, data) => {

if (err) throw err;

console.log(data);

});

**✅ 5. Path Module**

Helps to work with file paths.

const path = require('path');

console.log(path.basename(\_\_filename)); *// filename*

console.log(path.dirname(\_\_filename)); *// directory*

console.log(path.join(\_\_dirname, 'folder', 'file.txt'));

**✅ 6. Events Module**

Node is **event-driven**. Use EventEmitter to create and listen to events.

const EventEmitter = require('events');

const emitter = new EventEmitter();

emitter.on('greet', (name) => {

console.log(`Hello, ${name}`);

});

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

**✅ 7. OS Module**

Provides system-level info.

const os = require('os');

console.log(os.platform()); *// e.g., 'win32'*

console.log(os.cpus()); *// CPU info*

console.log(os.totalmem()); *// Total memory*

**✅ 8. process Object**

Gives access to system environment and runtime.

console.log(process.env); *// Environment variables*

console.log(process.argv); *// CLI arguments*

console.log(process.cwd()); *// Current directory*

Example with CLI:

node app.js hello

console.log(process.argv[2]); *// 'hello'*

**✅ 9. Streams and Buffers**

Used for large data, e.g., file/video streams.

const fs = require('fs');

const readStream = fs.createReadStream('file.txt', 'utf8');

readStream.on('data', chunk => {

console.log(chunk);

});

* **Stream**: Handle data piece-by-piece.
* **Buffer**: Temporary memory for binary data.

**✅ 10. CLI Tools in Node.js**

Build command-line apps using process.argv, fs, etc.

*// cli.js*

const name = process.argv[2];

console.log(`Hello, ${name}`);

node cli.js Alice

*# Output: Hello, Alice*

**🧠 Summary**

* Node modules help organize and reuse code.
* Built-in modules are powerful: fs, path, events, os, etc.
* CommonJS (require) is default; ES Modules (import) are modern.
* Use streams for performance and buffers for binary data.
* You can build command-line tools using core Node.js features.

**🔵 SECTION 1: Node.js – Basics to Advanced**

**📘 3. Asynchronous Programming in Node.js**

**✅ What is Asynchronous Programming?**

In Node.js, asynchronous programming is used to handle tasks **without blocking the main thread**. It allows your app to **handle multiple operations simultaneously**, like reading a file, querying a database, or calling an API—all **without waiting for each task to finish**.

**✅ 1. Callback Functions**

A **callback** is a function passed as an argument to another function, to be executed later (after async work is done).

**Example:**

const fs = require('fs');

fs.readFile('file.txt', 'utf8', (err, data) => {

if (err) return console.error(err);

console.log(data);

});

**❗ Callback Hell**

When callbacks are nested within callbacks, it becomes hard to read and maintain:

doSomething(() => {

doSomethingElse(() => {

yetAnotherThing(() => {

*// 😵 deeply nested!*

});

});

});

**✅ 2. Promises**

A **Promise** is an object representing the **future completion (or failure)** of an asynchronous operation.

**States:**

* pending
* fulfilled
* rejected

**Basic Usage:**

const fetchData = () => {

return new Promise((resolve, reject) => {

setTimeout(() => resolve("Data received"), 1000);

});

};

fetchData()

.then(data => console.log(data))

.catch(err => console.error(err));

**✅ 3. async / await**

Introduced in ES2017, async/await allows writing **async code that looks like synchronous code**. It’s built on top of Promises.

**Example:**

const fetchData = () => {

return new Promise((resolve) => {

setTimeout(() => resolve("Data received"), 1000);

});

};

async function main() {

try {

const result = await fetchData();

console.log(result);

} catch (err) {

console.error(err);

}

}

main();

✅ await can only be used inside async functions.

**✅ 4. Event Loop & Call Stack**

The **event loop** is what allows Node.js to perform non-blocking I/O operations.

**How it works:**

1. JavaScript runs in a **single thread**.
2. The **Call Stack** handles execution.
3. Async tasks (e.g. file read, API calls) go to the **Web APIs / Node APIs**.
4. When complete, callbacks are queued in the **Callback Queue**.
5. The **Event Loop** checks if the Call Stack is empty, then pushes queued callbacks into it.

🖼️ *(Use a diagram: Call Stack ⇄ Event Loop ⇄ Callback Queue)*

**✅ 5. Error Handling in Async Code**

**a. Callbacks:**

Always check err first:

fs.readFile('file.txt', 'utf8', (err, data) => {

if (err) return console.error(err);

console.log(data);

});

**b. Promises:**

Use .catch():

someAsyncFunction()

.then(result => console.log(result))

.catch(error => console.error(error));

**c. async/await:**

Use try...catch:

try {

const data = await fetchData();

} catch (err) {

console.error("Error occurred:", err);

}

**📝 Summary**

| **Concept** | **Use Case** |
| --- | --- |
| 🔁 Callbacks | Traditional way to handle async code |
| 📜 Promises | Modern way to chain async tasks |
| ⏱️ async/await | Cleaner, easier-to-read async code |
| 🔄 Event Loop | Powers non-blocking execution |
| ❗ Error Handling | Always handle async errors properly |

**🔵 SECTION 1: Node.js – Basics to Advanced**

**📘 4. HTTP & Server Basics**

**✅ 1. Creating a Basic HTTP Server**

Node.js comes with a built-in http module that allows us to create web servers without any external library.

**Example:**

const http = require('http');

const server = http.createServer((req, res) => {

res.end('Hello from Node.js Server!');

});

server.listen(3000, () => {

console.log('Server running at http://localhost:3000');

});

✅ This starts a server on port 3000 and sends a plain text response.

**✅ 2. Handling Requests and Responses**

You can read data from the request (req) and write to the response (res).

**Example: Basic routing**

const server = http.createServer((req, res) => {

if (req.url === '/') {

res.writeHead(200, {'Content-Type': 'text/plain'});

res.end('Home Page');

} else if (req.url === '/about') {

res.writeHead(200, {'Content-Type': 'text/plain'});

res.end('About Page');

} else {

res.writeHead(404, {'Content-Type': 'text/plain'});

res.end('404 Not Found');

}

});

🧠 Use req.url to route, and res.writeHead() to send headers (like content type, status code).

**✅ 3. Status Codes**

HTTP status codes are 3-digit numbers that indicate the result of an HTTP request.

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

**Example:**

res.writeHead(200, { 'Content-Type': 'text/plain' });

**✅ 4. Serving HTML and JSON**

**a. Serving HTML:**

const fs = require('fs');

const server = http.createServer((req, res) => {

if (req.url === '/') {

fs.readFile('index.html', (err, data) => {

res.writeHead(200, { 'Content-Type': 'text/html' });

res.end(data);

});

}

});

📁 Place index.html in the same folder.

**b. Serving JSON:**

const server = http.createServer((req, res) => {

if (req.url === '/api') {

const data = { name: 'John', age: 25 };

res.writeHead(200, { 'Content-Type': 'application/json' });

res.end(JSON.stringify(data));

}

});

**🧠 Summary**

| **Feature** | **Description** |
| --- | --- |
| http.createServer() | Creates a basic web server |
| req & res | Handle incoming request & send back response |
| writeHead() | Set HTTP status code and headers |
| Serve HTML | Use fs.readFile() and send with 'text/html' |
| Serve JSON | Use JSON.stringify() and 'application/json' |

**🔵 SECTION 1: Node.js – Basics to Advanced**

**📘 5. npm & Package Management**

**✅ 1. npm – Node Package Manager**

* **npm** is the default package manager for Node.js.
* Helps you **install**, **manage**, and **share** JavaScript packages/libraries.
* Comes bundled with Node.js.

Check version:

npm -v

**✅ 2. package.json & package-lock.json**

**a. package.json**

It is the **metadata file** of your Node.js project.

Generated using:

npm init *# Interactive*

npm init -y *# Default options*

**Example:**

{

"name": "my-app",

"version": "1.0.0",

"scripts": {

"start": "node app.js"

},

"dependencies": {

"express": "^4.18.2"

}

}

🔸 Contains info about project, scripts, dependencies, etc.

**b. package-lock.json**

* Automatically created when installing packages.
* Locks the exact version of every installed package and its sub-dependencies.
* Ensures **same versions** are installed across environments.

**✅ 3. Installing Packages**

**a. Install a package:**

npm install express

Installs package and adds it to dependencies in package.json.

**b. Install a specific version:**

npm install express@4.17.1

**c. Install globally:**

npm install -g nodemon

📦 Global packages are available from anywhere on your system.

**✅ 4. Dev Dependencies vs Prod Dependencies**

| **Type** | **Command** | **Stored In** | **Use For** |
| --- | --- | --- | --- |
| Dependencies | npm install <pkg> | "dependencies" | Required in production |
| Dev Dependencies | npm install <pkg> --save-dev | "devDependencies" | Only needed during development (e.g., testing, linting) |

**Example:**

npm install nodemon --save-dev

**✅ 5. Versioning (^, ~, etc.)**

| **Symbol** | **Meaning** | **Example** | **Version Range** |
| --- | --- | --- | --- |
| ^ | Accept latest minor/patch | ^1.2.3 | >=1.2.3 <2.0.0 |
| ~ | Accept latest patch | ~1.2.3 | >=1.2.3 <1.3.0 |
| No symbol | Exact version | 1.2.3 | =1.2.3 |

**Best Practice:**

* Use ^ for libraries you trust to follow **semver** (Semantic Versioning).
* Use exact versions in production if stability is critical.

**🧠 Summary**

| **Topic** | **Key Point** |
| --- | --- |
| package.json | Project metadata and dependency list |
| package-lock.json | Locks exact versions of installed packages |
| npm install | Installs dependencies |
| Dev vs Prod | --save-dev adds dev-only packages |
| Versioning | ^ = minor/patch updates, ~ = patch only |

**🔵 SECTION 1: Node.js – Basics to Advanced**

**📘 6. Debugging & Logging**

**✅ 1. console.log() – Basic Debugging**

The simplest way to debug:

console.log("Server started");

console.log({ req });

✅ Use it for:

* Quick variable inspection
* Printing function call flow
* Checking API responses

**✅ 2. debug Module**

Better alternative to console.log() for production-safe debugging.

**Install:**

npm install debug

**Use:**

const debug = require('debug')('app:server');

debug('Starting server...');

app.listen(3000, () => {

debug('Listening on port 3000');

});

🔹 Toggle output by setting the environment variable:

DEBUG=app:\* node app.js

**✅ 3. node inspect – Built-in Debugger**

Start your Node app in debug mode:

node inspect app.js

Use commands like:

* c → continue
* n → next line
* s → step into
* repl → open interactive console

🔹 Open Chrome DevTools:

node --inspect app.js

Then go to: chrome://inspect

**✅ 4. VSCode Debugger**

1. Add a breakpoint in VSCode by clicking beside the line number.
2. Go to the **Run and Debug** panel.
3. Create launch.json or click **“Run”** with “Node.js” config.

**Sample launch.json:**

{

"type": "node",

"request": "launch",

"name": "Debug App",

"program": "${workspaceFolder}/app.js"

}

✅ Benefits:

* Visual breakpoints
* Call stack and variable watch
* Step-in, step-over execution

**✅ 5. Logging Best Practices**

| **Practice** | **Description** |
| --- | --- |
| ✅ Use logging libraries | Use winston, morgan, or pino for structured logging |
| ✅ Separate logs | Separate logs for dev and production |
| ✅ Use log levels | e.g., info, warn, error, debug |
| ❌ Avoid logs in production | Avoid sensitive info in production logs |
| ✅ Store logs | Use files or external services (Logstash, Cloudwatch) |

**🔸 Exercises**

**✅ 1. Create a File Reader**

const fs = require('fs');

fs.readFile('data.txt', 'utf8', (err, data) => {

if (err) return console.error(err);

console.log(data);

});

**✅ 2. Basic HTTP Server**

const http = require('http');

const server = http.createServer((req, res) => {

res.end('Server running...');

});

server.listen(3000, () => console.log('Listening on 3000'));

**🔸 Mini Project: Simple File Upload Server**

**1. Setup**

Install express and multer:

npm install express multer

**2. upload.js**

const express = require('express');

const multer = require('multer');

const app = express();

const storage = multer.diskStorage({

destination: './uploads/',

filename: (req, file, cb) => {

cb(null, file.originalname);

}

});

const upload = multer({ storage });

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

res.send('File uploaded');

});

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

**3. Use Postman or frontend form to test /upload.**

**🔸 Interview Question**

**❓ What is event-driven programming in Node.js?**

**Answer:** Node.js uses **event-driven architecture**, meaning the flow of the program is determined by **events** (like requests, responses, I/O completion) and **callbacks** that respond to those events.

The core component is the **Event Loop**, which continuously listens for and dispatches events. This allows Node.js to handle **non-blocking I/O** and scale efficiently.

**Example:**

const EventEmitter = require('events');

const emitter = new EventEmitter();

emitter.on('message', (data) => {

console.log('Received:', data);

});

emitter.emit('message', 'Hello Event!');

**✅ Summary**

| **Tool** | **Use** |
| --- | --- |
| console.log() | Quick checks |
| debug | Namespace-based logging |
| node inspect | CLI debugging |
| VSCode Debugger | Visual breakpoints and step debugging |
| Logging tools | Use winston, pino, etc. for real projects |
|  |  |