Node.js CheatSheet

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

1. INTRODUCTION TO NODE.JS

1.1 What is Node.js?

- Node.js is a tool that helps you run JavaScript outside your browser.
- Think of it like this: normally JavaScript works inside websites (like in Google Chrome), but Node.js lets you run it like a regular computer program.

In short:

- JavaScript is for websites, Node.js is for building tools, servers, and apps using JavaScript.

1.2 Features of Node.js

Here are some special things about Node.js:

- Fast – It runs code very quickly.
- Non-blocking – It can handle many tasks at once.
- Uses JavaScript – No need to learn a new language.
- Cross-platform – Works on Windows, Mac, and Linux.
- Big Community – Many people use it and help each other.

1.3 Node.js vs Traditional Backend

Feature	Node.js	Traditional Backend (e.g., PHP, Java)
Language	JavaScript	PHP, Java, Python, etc.
Speed	Very Fast (non-blocking)	Slower (blocking by default)
Learning Curve	Easy for JS users	May need to learn new language
Use in Frontend	Same language as frontend	Usually different languages

In simple terms:

- Node.js is good if you already know JavaScript. It's fast and modern.

1.4 Use Cases

Where can we use Node.js?

- Web Servers – Like making your own version of Google or YouTube backend.
- APIs – Connecting front and back of a website.
- Real-time Apps – Like chatting or live games.
- Tools – Making tools to help developers.
- Command Line Apps – Programs that run in the terminal.

2. ENVIRONMENT SETUP

2.1 Installing Node.js & npm

- Go to https://nodejs.org
- Click "LTS" (Long Term Support) version
- Install it like any other software
- It will install Node.js and npm (Node Package Manager)

To check if it's installed:

```bash
node -v
npm -v
```

- This shows the version numbers.

2.2 REPL (Read-Eval-Print Loop)

- REPL is like a playground for trying Node.js code.

To open it:

1. Open Terminal or Command Prompt
2. Type node
3. You can now write JavaScript and press Enter

Example:

```js
> 2 + 2
4
```

- To exit REPL, type .exit and press Enter.

2.3 Creating Your First Script

1. Open a code editor (like VS Code)
2. Make a file: app.js
3. Write this:

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

4. Run it using terminal:

```bash
node app.js
```

- You'll see: Hello, Node.js!

2. ENVIRONMENT SETUP

2.4 Node.js in VS Code

Steps:

1. Download VS Code from https://code.visualstudio.com
2. Open your folder
3. Make .js files
4. Use terminal in VS Code (Ctrl + ` key)
5. Run code using node filename.js

- It’s an easy way to write, edit, and run your Node.js code in one place.

3. CORE MODULES IN NODE.JS

Core modules are tools that come built-in with Node.js. You don't need to install them — just use them.

3.1 fs – File System

- The fs module helps you read and write files.

Example:

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

fs.writeFileSync('hello.txt', 'This is Node.js');
```

- This will create a file called hello.txt.

3.2 path – Path Handling

- The path module helps you work with file and folder paths.

Example:

```js
const path = require('path');

const filePath = path.join(__dirname, 'folder', 'file.txt');
console.log(filePath);
```

- It joins paths safely across all computers.

3.3 http – Creating Servers

- This module lets you make your own web server.

Example:

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

const server = http.createServer((req, res) => {
  res.end('Hello from server');
});

server.listen(3000);
```

- Now visit : http://localhost:3000

3.4 url – URL Parsing

- The url module breaks a website link into parts.

Example:

```js
const url = require('url');

const parsed = url.parse('https://example.com/product?id=100');
console.log(parsed.query);
```

- It shows parts like path, query, host, etc.

3.5 os – Operating System Info

- Gives details about your computer’s system.

Example:

```js
const os = require('os');

console.log(os.platform());
console.log(os.freemem());
```

- Tells which OS you're using and free memory.

3.6 events – Event Handling

- Let’s you listen and respond to events.

Example:

```js
const EventEmitter = require('events');

const emitter = new EventEmitter();

emitter.on('start', () => {
  console.log('Started!');
});

emitter.emit('start');
```

- It’s like saying: “When I say START, do this.”

3.7 stream – Stream API

- Used to handle large files like videos, step-by-step.

Example:

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

const readStream = fs.createReadStream('bigfile.txt');

readStream.on('data', chunk => {
  console.log('Reading chunk:', chunk);
});
```

- Reads file in parts, not all at once.

4. NPM (NODE PACKAGE MANAGER)

NPM helps you install ready-made code from others. These codes are called packages or modules.

4.1 What is npm?

- NPM stands for Node Package Manager.
- It comes with Node.js.
- It helps you download packages to add features quickly.

4.2 Installing Packages (npm install)

- To install a package:

```bash
npm install package-name
```

- Example:

```bash
npm install chalk
```

- This downloads and saves the package in your project folder.

4.3 Local vs Global Packages

- Local Package – Used only in your project

```bash
npm install chalk
```

- Global Package – Used in all projects

```bash
npm install -g nodemon
```

- Global means "install it on your computer, not just project"

4.4 package.json and package-lock.json

- package.json – Keeps list of your installed packages
- It's like your project's "ingredients list"

```json
{
  "name": "my-app",
  "dependencies": {
    "chalk": "^5.0.0"
  }
}
```

- package-lock.json – Keeps exact version details of packages
- More like a detailed recipe

4. NPM (NODE PACKAGE MANAGER)

4.5 Semantic Versioning

- Package versions use 3 numbers:
- major.minor.patch

Example: 1.2.3

- 1 → Big changes
- 2 → New features
- 3 → Bug fixes only

Symbols:

- ^ → Allow minor and patch updates
- ~ → Allow only patch updates

5. MODULES IN NODE.JS

In real life, when you work on a big project, you don't write everything in one place.

- You break it into small files.
- These files are called modules in Node.js.

Node.js has two types of modules:

- CommonJS (Old style)
- ES Modules (New style)

5.1 What are CommonJS Modules?

- CommonJS is the default module system in Node.js.

Two main parts:

- require() – to import a module
- module.exports – to export from a module

Example:

math.js

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

module.exports = add;
```

app.js

```js
const add = require('./math');
console.log(add(5, 3)); // Output: 8
```

How it works:

- You write a function inside math.js
- You export it using module.exports
- You import it in app.js using require()

This is like making a toy in one room and using it in another room.

5.2 What are ES Modules?

- ES Modules use the modern JavaScript syntax:
- import and export.

To use ES Modules:

- Your file should end with .mjs
- OR
- Add "type": "module" in package.json

5. MODULES IN NODE.JS

5.2 What are ES Modules?

- Example:

math.mjs

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

app.mjs

```js
import { add } from './math.mjs';
console.log(add(10, 2)); // Output: 12
```

- Use ES Modules if you're using the latest JavaScript features.

5.3 Creating and Reusing Your Own Modules

- You can create any number of modules.

Example project:

- math.js – contains math functions
- user.js – contains user data
- server.js – your main file

math.js

```js
function multiply(a, b) {
  return a * b;
}

module.exports = multiply;
```

server.js

```js
const multiply = require('./math');
console.log(multiply(4, 5)); // Output: 20
```

Why use modules?

- Easy to organize
- Easy to reuse
- Code looks clean and neat

6. ASYNCHRONOUS PROGRAMMING

- Node.js does many things at the same time.
- It doesn't wait for one task to finish.
- This is called asynchronous programming.

6.1 Callbacks (Old Method)

- A callback is a function you pass into another function.
- It gets called after the first function finishes.

Example:

```js
function greet(name, callback) {
  console.log("Hi " + name);
  callback();
}

greet("Ravi", function () {
  console.log("Welcome to Node.js!");
});
```

Problem:

- If you have many callbacks inside callbacks, your code looks messy.
- That's called callback hell.

6.2 Promises (Better Way)

- A promise is like saying:
- "I promise I'll finish this task. If it works, I'll give you the result. If not, I'll give an error."

Creating a Promise

```js
const promise = new Promise((resolve, reject) => {
  let success = true;

  if (success) {
    resolve("It worked!");
  } else {
    reject("It failed!");
  }
});

promise
  .then(result => console.log(result))
  .catch(error => console.log(error));
```

- .then() → when it works
- .catch() → when it fails

6. ASYNCHRONOUS PROGRAMMING

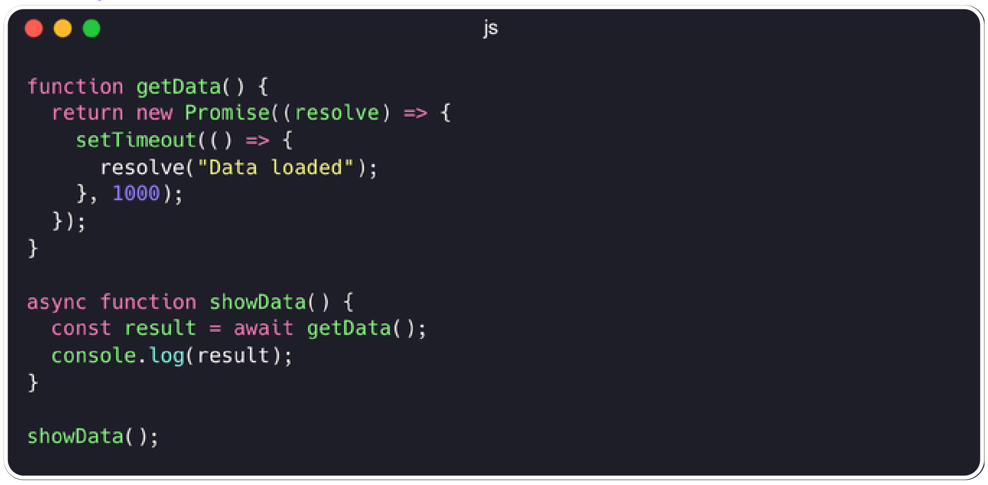
6.3 async/await (Modern Way)

- async/await makes your code look like normal code, but it works asynchronously.

You use:

- async before a function
- await before a promise

Example:



```js
function getData() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("Data loaded");
    }, 1000);
  });
}

async function showData() {
  const result = await getData();
  console.log(result);
}

showData();
```

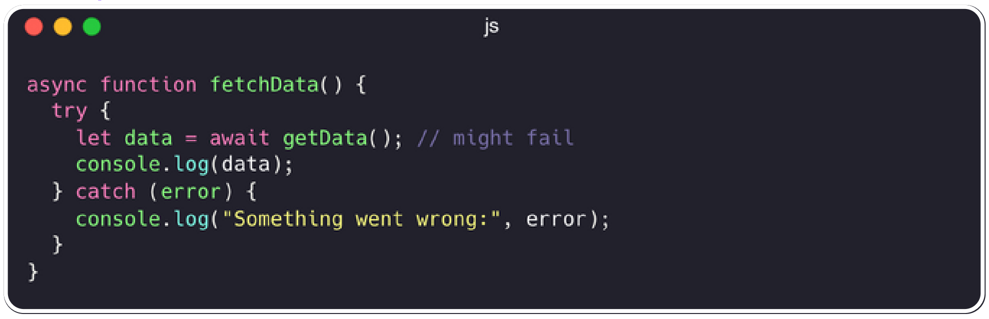
Result:

- After 1 second → "Data loaded"
- It’s easier to read and manage than promises or callbacks.

6.4 Error Handling in async/await

- Always use try...catch to handle errors in async functions.

Example:



```js
async function fetchData() {
  try {
    let data = await getData(); // might fail
    console.log(data);
  } catch (error) {
    console.log("Something went wrong:", error);
  }
}
```

- This stops your app from crashing.

7. FILE SYSTEM OPERATIONS (WITH FS MODULE)

- To work with files (like reading or writing text files), Node.js gives us the fs module.
- It helps us handle documents and folders.

7.1 Reading/Writing Files (Sync vs Async)

You can read/write files in two ways:

- Sync (Synchronous) – Waits until file work is done
- Async (Asynchronous) – Does other tasks while file work runs

Example (Sync):

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

const data = fs.readFileSync('file.txt', 'utf-8');
console.log(data);
```

- This will stop other code until reading is done.

Example (Async):

```js
fs.readFile('file.txt', 'utf-8', (err, data) => {
  if (err) return console.log(err);
  console.log(data);
});
```

- This will read the file in the background and continue other tasks.

7.2 Creating/Deleting Directories

Create a Folder:

```js
fs.mkdir('myFolder', (err) => {
  if (err) throw err;
  console.log('Folder created');
});
```

Delete a Folder:

```js
fs.rmdir('myFolder', (err) => {
  if (err) throw err;
  console.log('Folder deleted');
});
```

7. FILE SYSTEM OPERATIONS (WITH FS MODULE)

7.3 File Streams

- Streams are used when files are very big.
- Instead of reading all at once, Node.js reads in small chunks.

Read Stream:

```js
const readStream = fs.createReadStream('bigfile.txt');

readStream.on('data', (chunk) => {
  console.log('Reading part:', chunk);
});
```

- This avoids memory overload.

8. HTTP SERVER

- Node.js has a built-in module called http.
- It helps us create a web server without any extra tools.

8.1 Creating a Simple Server

```js
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');
});
```

Explanation:

- http.createServer() creates a server
- res.end() sends a message to the browser
- listen(3000) means it runs on port 3000

8.2 Handling Requests and Responses

When someone visits your server, you can check:

- req.url → to know what page they want
- req.method → to know if they used GET, POST, etc.

```js
const server = http.createServer((req, res) => {
  if (req.url === '/about') {
    res.end('About Page');
  } else {
    res.end('Welcome Home');
  }
});
```

8.3 Routing Basics

- Routing means giving different responses based on the URL path.

Example:

```js
if (req.url === '/') {
  res.end('Home Page');
} else if (req.url === '/contact') {
  res.end('Contact Page');
} else {
  res.end('Page Not Found');
}
```

Note: This manual routing becomes hard in big apps — that's where Express.js helps.

8. HTTP SERVER

8.4 Sending JSON and HTML Responses

- You can send HTML or JSON as responses by setting the right content type.

Send HTML:

```js
res.setHeader('Content-Type', 'text/html');
res.end('<h1>Hello HTML Page</h1>');
```

Send JSON:

```js
res.setHeader('Content-Type', 'application/json');
res.end(JSON.stringify({ message: 'Hello JSON' }));
```

- This is how a server talks to the browser in different formats.

9. EXPRESS.JS

- Express.js is a small and fast framework built on top of Node.js.
- It helps create servers easily and quickly.

9.1 What is Express.js?

- Express is like a helper or shortcut for Node.js servers
- Makes routing, middleware, and file handling super simple
- Used in almost every real-world Node.js project

9.2 Setting Up Express App

- Step 1: Install Express

```bash
npm install express
```

- Step 2: Create a basic app

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

app.get('/', (req, res) => {
  res.send('Welcome to Express.js!');
});

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

CHECK OUT

Explanation:

- express() creates the app
- app.get() handles GET request
- res.send() sends text or HTML

9.3 Routing & Middleware

Routing

```js
app.get('/', (req, res) => {
  res.send('Home Page');
});

app.get('/about', (req, res) => {
  res.send('About Page');
});
```

- This is cleaner than http.createServer() routing.

Middleware

- Middleware is a function that runs before the route. It helps do tasks like logging, checking tokens, etc.

9. EXPRESS.JS

Middleware

- Middleware is a function that runs before the route. It helps do tasks like logging, checking tokens, etc.

js

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

9.4 Serving Static Files

- You can serve images, CSS, and JS files using one line:

js

```js
app.use(express.static('public'));
```

- Put your static files (like index.html, style.css) inside a folder named public.

Then:

- /logo.png → loads public/logo.png
- /style.css → loads public/style.css

9.5 Error Handling

- If someone visits a route that doesn't exist, show a custom error:

js

```js
app.use((req, res) => {
  res.status(404).send('404 Page Not Found');
});
```

- This middleware runs last, if no other route matches.

10. WORKING WITH DATABASES

- Node.js can connect to many databases.

We'll focus on:

- MongoDB → NoSQL (stores data like JSON)
- MySQL → SQL (stores data in tables)

We use a library called Mongoose to easily work with MongoDB.

10.1 Connecting to MongoDB using Mongoose

Step 1: Install Mongoose

```bash
npm install mongoose
```

Step 2: Connect to MongoDB

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

mongoose.connect('mongodb://localhost:27017/myDB')
  .then(() => console.log('MongoDB Connected'))
  .catch(err => console.log(err));
```

- This connects your app to MongoDB. Replace 'myDB' with your database name.

10.2 CRUD Operations (Create, Read, Update, Delete)

- Let's create a simple User model and do all four basic actions.

1. Create Schema

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

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

Create

```js
const newUser = new User({ name: 'Ravi', age: 20 });
newUser.save();
```

Read

```js
User.find().then(users => console.log(users));
```

Update

```js
User.updateOne({ name: 'Ravi' }, { age: 21 });
```

Delete

```js
User.deleteOne({ name: 'Ravi' });
```

10.3 MySQL with Node.js

- To work with MySQL, install this:

```bash
npm install mysql
```

- Connecting to MySQL:

```js
const mysql = require('mysql');

const db = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: '',
  database: 'testdb'
});

db.connect(err => {
  if (err) throw err;
  console.log('MySQL Connected');
});
```

Example Query:

```js
db.query('SELECT * FROM users', (err, result) => {
  if (err) throw err;
  console.log(result);
});
```

10.4 Using .env for Config

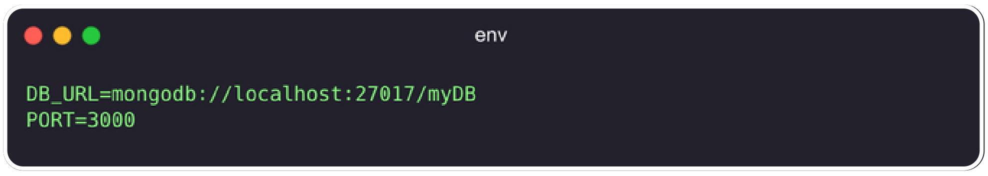
- We store secret data (like DB passwords) in a .env file.

Step 1: Install dotenv

```js
npm install dotenv
```

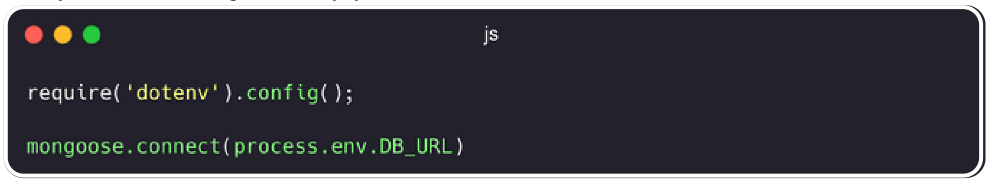
10. WORKING WITH DATABASES

Step 2: Create .env file



```
DB_URL=mongodb://localhost:27017/myDB
PORT=3000
```

Step 3: Use in your app



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

mongoose.connect(process.env.DB_URL)
```

- This keeps your sensitive info safe and hidden.

11. EVENT-DRIVEN ARCHITECTURE

- Node.js is built on events. It means:
- "When something happens, do something."
- Node gives us a tool called EventEmitter to make this easy.

11.1 EventEmitter in Action

Step 1: Import it

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

Step 2: Listen to an event

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

Step 3: Emit the event

```js
emitter.emit('greet');
```

Output:

```nginx
Hello World!
```

11.2 Custom Events

- You can send data too:

```js
emitter.on('userAdded', (name) => {
  console.log('User added:', name);
});

emitter.emit('userAdded', 'Amit');
```

11.3 Broadcasting Events

- You can trigger one event that causes others to run.

```js
emitter.on('start', () => {
  console.log('Starting...');
  emitter.emit('nextStep');
});

emitter.on('nextStep', () => {
  console.log('Doing next step...');
});

emitter.emit('start');
```

11. EVENT-DRIVEN ARCHITECTURE

11.3 Broadcasting Events

Output:

```vbnet
Starting...
Doing next step...
```

12. AUTHENTICATION & SECURITY IN NODE.JS

- When users sign up or log in, we must secure their passwords and control who can access what.

We use tools like:

- bcrypt → to hash (scramble) passwords
- JWT → to keep users logged in securely
- helmet.js → to protect the app from attacks
- rate-limiter → to stop spamming or too many requests

12.1 Hashing with bcrypt

- Passwords should never be stored directly.
- We hash them — this means we scramble them so no one can read them.

Step 1: Install bcrypt

```bash
npm install bcrypt
```

Step 2: Hash Password Before Saving

```js
const bcrypt = require('bcrypt');

const password = 'mypassword123';
bcrypt.hash(password, 10, (err, hash) => {
  console.log('Hashed Password:', hash);
});
```

Check Password on Login

```js
bcrypt.compare('mypassword123', hash, (err, result) => {
  if (result) {
    console.log('Password is correct');
  } else {
    console.log('Wrong password');
  }
});
```

12.2 JWT (JSON Web Tokens)

- JWT is used to keep users logged in after they log in once.
- It sends a secret token that proves the user is real.

Step 1: Install

```bash
npm install jsonwebtoken
```

Step 2: Sign (Create) a Token

```js
const jwt = require('jsonwebtoken');
const token = jwt.sign({ userId: 123 }, 'secretKey', { expiresIn: '1h' });
```

Step 3: Verify Token

```js
jwt.verify(token, 'secretKey', (err, data) => {
  if (err) console.log('Invalid token');
  else console.log('Valid user:', data);
});
```

- Use .env to hide the secret key: JWT_SECRET=secretKey

12.3 Helmet.js (Security Headers)

- Helmet helps protect the app from common attacks like cross-site scripting (XSS).

Step 1: Install

```bash
npm install helmet
```

Step 2: Use in app

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

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

12.4 Rate Limiting

- This stops users from sending too many requests (like login spam).

Step 1: Install

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

Step 2: Use in app

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

const limiter = rateLimit({
  windowMs: 1 * 60 * 1000, // 1 minute
  max: 5 // limit each IP to 5 requests per minute
});

app.use(limiter);
```

- This helps block bots or repeated spam clicks.

13. REAL-TIME COMMUNICATION

- Normal websites wait for the user to refresh the page to see updates.
- Real-time apps (like chat) show updates instantly.
- This is done using WebSockets and the socket.io library.

13.1 Using WebSockets with socket.io

- WebSockets help keep a live connection between the user and server.

Step 1: Install socket.io

```bash
npm install socket.io
```

Step 2: Create a Socket Server

```js
const express = require('express');
const http = require('http');
const socketIo = require('socket.io');

const app = express();
const server = http.createServer(app);
const io = socketIo(server);

io.on('connection', socket => {
  console.log('A user connected');

  socket.on('message', msg => {
    console.log('Message:', msg);
  });

  socket.on('disconnect', () => {
    console.log('User disconnected');
  });
});

server.listen(3000);
```

- Now users can send and receive messages instantly.

13.2 Real-Time Chat Example

- Frontend HTML (very simple):

```html
<script src="/socket.io/socket.io.js"></script>
<script>
  const socket = io();

  socket.emit('message', 'Hello from client');
  socket.on('message', data => {
    console.log('Got message:', data);
  });
</script>
```

- Backend (send message back to client):

```js
socket.on('message', msg => {
  io.emit('message', 'Server says: ' + msg);
});
```

- Now whatever one user types, others will see instantly.

13. REAL-TIME COMMUNICATION

13.3 Rooms & Broadcasting Events

- You can split users into rooms, like room1, room2 in chat apps.

Join Room:

```js
socket.join('room1');
```

Send Message to Room Only:

```js
io.to('room1').emit('message', 'Hello room1');
```

- Now only people in room1 get the message.
- This is helpful for group chats, game lobbies, etc.

14. ERROR HANDLING & DEBUGGING

- When building apps, things go wrong. We need to handle errors properly and debug to fix issues.

14.1 Try/Catch & next(err)

- Try/Catch — Handle code errors
- Use try to run code. If it fails, catch will catch the error.

```js
try {
  const data = JSON.parse('{ wrong json }');
} catch (err) {
  console.log('There was an error:', err.message);
}
```

- In Express, use next(err)
- It passes the error to a special error handler.

```js
app.get('/', (req, res, next) => {
  try {
    throw new Error('Something went wrong');
  } catch (err) {
    next(err); // passes error
  }
});

// Error handling middleware
app.use((err, req, res, next) => {
  res.status(500).send('Internal Server Error: ' + err.message);
});
```

14.2 Built-in Debugging Tools

Common ways to debug:

- console.log() – Print values to check
- debugger; – Pauses code when using the debugger
- Breakpoints – Pause at any line in VS Code

Example:

```js
let x = 5;
let y = 10;

debugger; // VS Code will pause here

let sum = x + y;
console.log(sum);
```

14. ERROR HANDLING & DEBUGGING

14.3 console.log() vs VS Code Debugging

Method	What it does	When to use
console.log()	Prints values in terminal	Small checks or logs
VS Code Debugger	Step-by-step code flow & watch values	Deeper investigation

- Both are useful. Start with console.log(), use debugger for tricky bugs.

15. BEST PRACTICES & FOLDER STRUCTURE

Keeping your project clean and organized helps in scaling, debugging, and team collaboration.

15.1 MVC Pattern in Node.js

MVC = Model, View, Controller

- Model → Deals with data (e.g., MongoDB)
- View → HTML or frontend files (optional in APIs)
- Controller → Logic and functions (routes, APIs)

Example Structure:

```arduino
project/
│
├── models/          → DB schema
├── controllers/     → Logic functions
├── routes/          → URL handling
├── views/           → Templates (optional)
├── config/          → DB, dotenv setup
└── app.js           → Entry point
```

15.2 Environment-Based Config

- Different settings for development vs production.
- Use .env file for secrets and environment values.

```env
PORT=3000
DB_URL=mongodb://localhost:27017/devDB
```

Access in app:

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

- Use .env.production for live server settings.

15.3 Modular Coding

- Break big files into smaller files.

Instead of writing all logic in app.js, split it:

- userController.js → handles user logic
- productController.js → handles products
- db.js → handles DB connection

Benefits:

- Easier to manage
- Reuse code
- Clean structure

15. BEST PRACTICES & FOLDER STRUCTURE

15.4 Avoiding Callback Hell

- Callback Hell = Too many nested functions like:

```js
doA(() => {
  doB(() => {
    doC(() => {
      doD(() => {
        // 😲 too deep
      });
    });
  });
});
```

- Use Promises or async/await instead:

```js
async function run() {
  await doA();
  await doB();
  await doC();
}
```

16. TESTING IN NODE.JS

- Testing helps make sure your code works correctly before using it in real apps.
- There are 3 main types of testing in Node.js:

16.1 Unit Testing with Mocha & Chai

- Unit Testing checks small parts of your app (like one function).
- Mocha is a test runner (it runs your tests).
- Chai is an assertion tool (it checks if results are correct).

Step 1: Install Mocha & Chai

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

Step 2: Create a function to test (math.js)

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

Step 3: Write test file (test/math.test.js)

```js
const add = require('../math');
const chai = require('chai');
const expect = chai.expect;

describe('Addition', () => {
  it('should return 5 for 2 + 3', () => {
    expect(add(2, 3)).to.equal(5);
  });
});
```

Step 4: Add test script in package.json

```json
"scripts": {
  "test": "mocha"
}
```

Step 5: Run tests

```bash
npm test
```

- If your function works, the test will pass. Otherwise, it will show an error.

16. TESTING IN NODE.JS

16.2 Integration Testing

- Integration testing checks if multiple parts work together.
- Example: You test if a user can register and get a response from the API.

Tools:

- supertest to test HTTP endpoints
- Combine with Mocha & Chai

```bash
npm install --save-dev supertest
```

Example:

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

describe('GET /home', () => {
  it('should return 200 OK', async () => {
    const res = await request(app).get('/home');
    expect(res.status).to.equal(200);
  });
});
```

16.3 Test Coverage

- Test coverage shows how much of your code is tested.
- Tool: nyc (Istanbul)

```bash
npm install --save-dev nyc
```

- Update package.json:

```json
"scripts": {
  "test": "mocha",
  "coverage": "nyc npm test"
}
```

- Run:

```bash
npm run coverage
```

- It will tell you what % of your code is covered by tests.

17. DEPLOYMENT IN NODE.JS

Once your app is ready, you can deploy it — which means put it online for people to use.

17.1 Using pm2 for Production

- pm2 keeps your app running forever, even if it crashes.
- It also helps restart the app automatically.

Step 1: Install pm2

```bash
npm install -g pm2
```

Step 2: Run your app

```bash
pm2 start app.js
```

Step 3: Save process list

```bash
pm2 save
```

Step 4: Startup script (auto-start on reboot)

```bash
pm2 startup
```

- Great for Linux VPS servers like AWS EC2, DigitalOcean, etc.

17.2 Hosting on Heroku, Vercel, or Render

- You can host your Node.js app online for free or low cost.

Heroku (good for beginners)

1. Create Heroku account
2. Install Heroku CLI
3. Initialize git in your project:

```bash
git init
heroku login
heroku create your-app-name
git add .
git commit -m "first commit"
git push heroku master
```

- Done! App is live.

17. DEPLOYMENT IN NODE.JS

Vercel or Render

- Vercel is good for frontends, but Render works great with full Node.js backend.
- Go to https://render.com/
- Connect GitHub
- Select repo
- Set start command: node app.js
- Add environment variables if needed
- Deploy!

Render offers auto-deploy when you push to GitHub.

17.3 Environment Variables in Production

- Your .env file should not be uploaded. Use secrets in your hosting dashboard.

Example: “On Heroku:”

```bash
heroku config:set DB_URL=mongodb+srv://user:pass@cluster
```

In your app:

```js
const db = process.env.DB_URL;
```

- This keeps sensitive data safe and hidden.

18. POPULAR NODE.JS PACKAGES

Node.js has many helpful packages that make coding easier and faster. Here are some must-know ones for beginners:

18.1 nodemon – Auto Restart

- Restarts your server automatically when code changes.
- Saves time in development.

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

In package.json:

```json
"scripts": {
  "start": "node app.js",
  "dev": "nodemon app.js"
}
```

Run:

```bash
npm run dev
```

18.2 dotenv – Manage Secrets

- Loads .env file so you can hide API keys, DB URLs, etc.

```bash
npm install dotenv
```

Create .env:

```ini
PORT=3000
```

Use in your app:

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

18.3 cors – Fix Cross-Origin Errors

- Lets your frontend talk to your backend if they are on different domains or ports.

```bash
npm install cors
```

18. POPULAR NODE.JS PACKAGES

18.3 cors – Fix Cross-Origin Errors

- use:

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

18.4 axios – Make HTTP Requests

- Helps you make API calls (GET, POST, etc.).

```bash
npm install axios
```

Use:

```js
const axios = require('axios');
axios.get('https://api.example.com/data')
  .then(res => console.log(res.data));
```

18.5 express-validator – Validate Inputs

- Checks if user inputs are correct (like email, password).

```bash
npm install express-validator
```

Use:

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

app.post('/register', [
  body('email').isEmail(),
  body('password').isLength({ min: 5 })
], (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) return res.status(400).json({ errors: errors.array()
});
  res.send('Valid!');
});
```

18.6 mongoose – Work with MongoDB

- Makes MongoDB easier to use with Node.js

```bash
npm install mongoose
```

Use:

```js
const mongoose = require('mongoose');
mongoose.connect(process.env.DB_URL);
```

18. POPULAR NODE.JS PACKAGES

18.7 jsonwebtoken – Auth with Tokens

- Creates and verifies login tokens (JWTs).

```bash
npm install jsonwebtoken
```

Use:

```js
const jwt = require('jsonwebtoken');
const token = jwt.sign({ id: 1 }, 'secret');
const decoded = jwt.verify(token, 'secret');
```

18.8 multer – File Uploads

- Upload images, documents, etc.

```bash
npm install multer
```

Use:

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

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

18.9 chalk – Colorful Logs

- Makes terminal messages colorful.

```bash
npm install chalk
```

Use:

```js
const chalk = require('chalk');
console.log(chalk.green('Success!'));
```

18.10 uuid – Unique IDs

- Generate unique IDs for users, files, etc.

```bash
npm install uuid
```

Use:

```js
const { v4: uuidv4 } = require('uuid');
console.log(uuidv4()); // random ID
```

19. COMMON BEGINNER MISTAKES

Avoid these mistakes to become a better Node.js developer:

19.1 Blocking the Event Loop

- ❌ Writing code that waits too long blocks Node.js from doing other tasks.

Bad:

```js
while(true) {
  // never ends — blocks everything
}
```

✅ Use non-blocking code like:

```js
setTimeout(() => {
  console.log('Run later');
}, 1000);
```

19.2 Ignoring Error Handling

- ❌ Not using try/catch or .catch() can crash your app.

Bad:

```js
const fs = require('fs');
fs.readFileSync('not_found.txt'); // app crashes if file not found
```

Good:

```js
try {
  const data = fs.readFileSync('file.txt');
} catch (err) {
  console.log('Error:', err.message);
}
```

Or with Promises:

```js
axios.get('url')
  .then(res => console.log(res))
  .catch(err => console.log('API error'));
```

19. COMMON BEGINNER MISTAKES

19.3 Improper Module Imports

- ❌ Mixing CommonJS and ES Modules incorrectly.

Bad:

```js
import fs from 'fs'; // ❌ if using require-based project
```

Use one format consistently:

```js
const fs = require('fs'); // CommonJS
```

Or if using ES Modules (with "type": "module" in package.json):

```js
import fs from 'fs';
```

19.4 Hardcoding Sensitive Info

❌ Storing passwords, API keys in code.

Bad:

```js
const dbURL = 'mongodb+srv://user:pass@cluster.mongodb.net';
```

Good:

```env
# .env
DB_URL=mongodb+srv://user:pass@cluster
```

```js
require('dotenv').config();
const dbURL = process.env.DB_URL;
```

- Always use .env and never upload it to GitHub.

SUMMARY TABLE

No.	Topic	What It Covers (Simple Summary)
1	Introduction to Node.js	What Node.js is, its features, how it's different, and what you can build with it.
2	Environment Setup	Installing Node.js, npm, using REPL, writing your first script, using VS Code.
3	Core Modules	Built-in tools like fs, http, path, url, events, os, stream modules.
4	NPM (Node Package Manager)	Installing packages, using package.json, local vs global, semantic versioning.
5	Modules in Node.js	Import/export with CommonJS and ES Modules, reusable custom modules.
6	Asynchronous Programming	Callbacks, Promises, async/await, and handling errors in async code.
7	File System Operations	Reading/writing files, creating/deleting folders, using streams for large files.
8	HTTP Server	Creating a server, handling requests/responses, basic routing, sending HTML/JSON.
9	Express.js	What Express is, setting it up, routes, middleware, static files, and error handling.
10	Working with Databases	Connecting MongoDB with Mongoose, basic CRUD, MySQL support, using .env for configs.
11	Event-Driven Architecture	Using EventEmitter, creating custom events, and broadcasting them.
12	Authentication & Security	Password hashing (bcrypt), JWTs, Helmet.js for safety, rate limiting for protection.
13	Real-Time Communication	Using socket.io for WebSockets, building live chat, using rooms and event broadcasting.
14	Error Handling & Debugging	Try/catch, error middleware, VS Code debugging, console.log() vs tools.
15	Best Practices & Folder Structure	MVC pattern, clean modular code, environment-based configs, avoiding callback hell.
16	Testing in Node.js	Unit tests (Mocha/Chai), integration tests, test coverage using nyc.
17	Deployment	Using pm2, hosting on Heroku/Render, managing environment variables in production.
18	Popular Node.js Packages	Useful npm packages like nodemon, dotenv, cors, axios, multer, uuid, and more.
19	Common Beginner Mistakes	Blocking the event loop, no error handling, mixing imports, hardcoding secrets.

Was this post helpful ?

coders_section

Coding | Programming | HTML | CSS

130 posts 17K followers 92 following

Follow Our 2nd Account

coders.100

Coding • HTML • CSS • JAVASCRIPT

5 posts 31 followers 4 following

Follow For More

Tap Here