upGrad ENTERPRISE

Node.js Advanced + HTTP Server — Day 2

Siddhesh Prabhugaonkar
Microsoft Certified Trainer
siddheshpg@azureauthority.in

- **Day 1 Recap:**
  - Introduction to Node.js and its architecture
  - Event loop & asynchronous programming
  - Core modules (fs, path, http, etc.)
  - npm & package.json basics
  - Writing and running simple Node.js scripts
- **Day 2 Agenda:**
  - Modules & require/import
  - Callbacks, Promises, and async/await
  - Error handling in Node.js
  - Working with the filesystem (CRUD)
  - Streams & buffers basics

Modules & require/import

- Node.js uses the **CommonJS module system** (`require`) and also supports **ES Modules** (`import/export`).
- Helps organize code into smaller, reusable files.

Example – CommonJS:

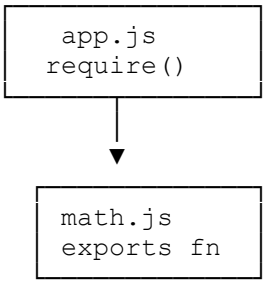
```
// math.js
function add(a, b) {
  return a + b;
}
module.exports = add;

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

Example – ES Modules (modern style):

```
// math.mjs
export const multiply = (a, b) => a * b;

// app.mjs
import { multiply } from './math.mjs';
console.log(multiply(4, 5));
```

Tip:
Use `"type": "module"` in `package.json` to enable ES modules in `.js` files.

Module Resolution:



Activity:
Create a custom `utils.js` that exports functions for addition and greeting and import it into `index.js`.

Step 1 - Create `utils.js`

```
// utils.js

// A simple function to add two numbers
function add(a, b) {
  return a + b;
}

// A function to greet a user
function greet(name) {
  return `Hello, ${name}! Welcome to Node.js`;
}

// Export the functions so they can be used in other files
module.exports = {
  add,
  greet,
};
```

Step 2 – Create `index.js`

```
// index.js

// Import functions from utils.js
const { add, greet } = require('./utils');

// Use the functions
const sum = add(10, 20);
console.log(`Sum: ${sum}`);

const message = greet('Siddhesh');
console.log(message);
```

Run:

```
node index.js
```

Expected Output:

```
Sum: 30
Hello, Siddhesh! Welcome to Node.js
```

Variation – ES Modules version (Optional Modern Syntax)

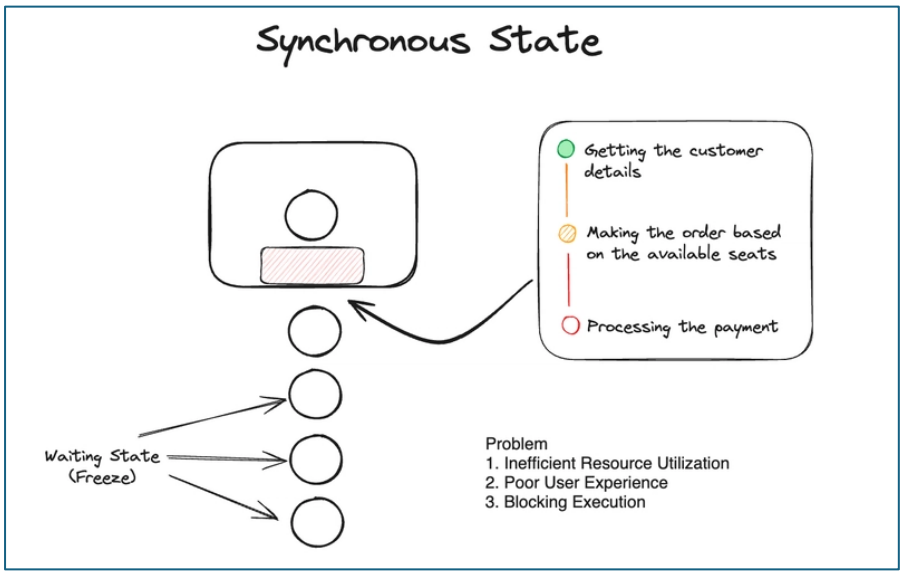
If `package.json` includes `"type": "module"`, use `import/export`:

```
// utils.mjs
export function add(a, b) {
  return a + b;
}

export function greet(name) {
  return `Hello, ${name}!`;
}
// index.mjs
import { add, greet } from './utils.mjs';

console.log(add(5, 7));
console.log(greet('Student'));
```

Run:

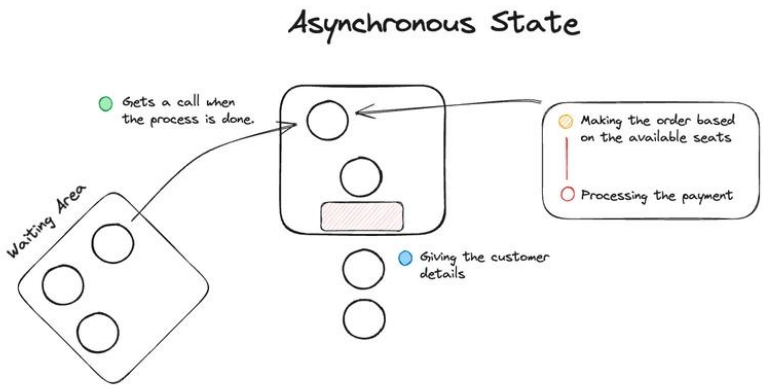
```
node index.mjs
```

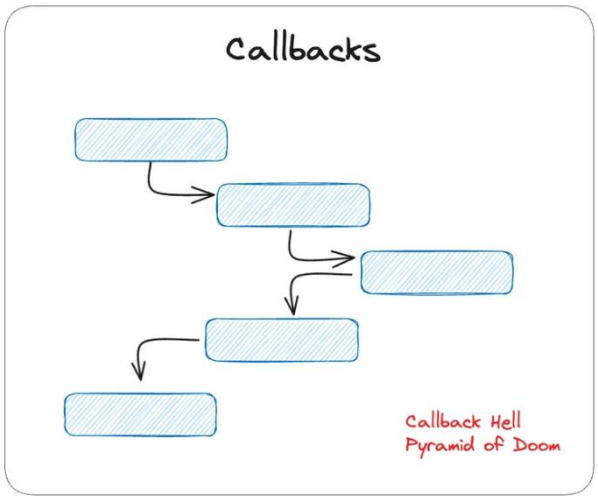
Callbacks, Promises, and async/await

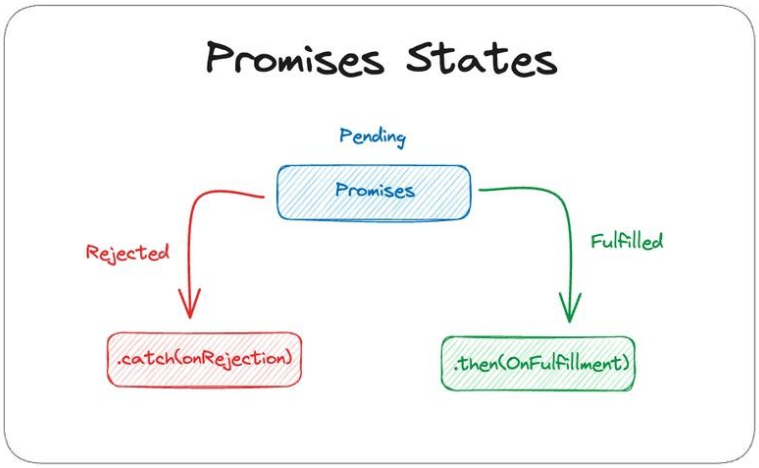


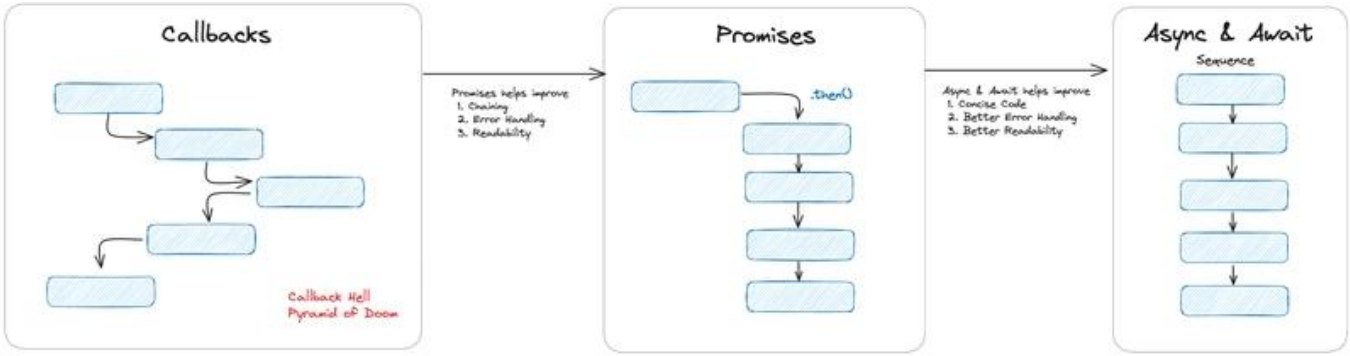
- Asynchronous operations can be handled in 3 main ways:
    1. **Callbacks** (older style)
    2. **Promises**
    3. **async/await** (simpler syntax for promises)

Callbacks









Callback Example:

```
function getData(callback) {
  setTimeout(() => {
    callback("Data fetched");
  }, 1000);
}

getData((msg) => console.log(msg));
```

Promise Example:

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

getData().then(console.log);
```

async/await Example:

```
async function fetchData() {
  const msg = await getData();
  console.log(msg);
}
fetchData();
```

Activity:
Rewrite a callback-based function using Promises and then convert it to `async/await`.

Step 1 – Start with a Callback Version

```
// callback-demo.js

function fetchData(callback) {
  console.log("Fetching data...");
  setTimeout(() => {
    callback(null, "Data fetched successfully!");
  }, 1500);
}

fetchData((err, result) => {
  if (err) {
    console.error("Error:", err);
  } else {
    console.log(result);
  }
});
```

Output:

```
Fetching data...
Data fetched successfully!
```

Step 2 – Rewrite using Promises

```
// promise-demo.js

function fetchData() {
  return new Promise((resolve, reject) => {
    console.log("Fetching data...");
    setTimeout(() => {
      // Simulate success
      resolve("Data fetched successfully!");
      // Uncomment to test error: reject("Failed to fetch data!");
    }, 1500);
  });
}

// Using then() and catch()
fetchData()
  .then((result) => console.log(result))
  .catch((err) => console.error("Error:", err));
```

Output:

```
Fetching data...
Data fetched successfully!
```

Step 3 – Rewrite using async/await

```
// async-await-demo.js

function fetchData() {
  return new Promise((resolve) => {
    console.log("Fetching data...");
    setTimeout(() => resolve("Data fetched successfully!"), 1500);
  });
}

async function run() {
  try {
    const result = await fetchData();
    console.log(result);
  } catch (err) {
    console.error("Error:", err);
  }
}

run();
```

Output:

```
Fetching data...
Data fetched successfully!
```

Add an intentional error (`reject("Error!")`) to demonstrate `try...catch`.

Tip:
Use the Chrome DevTools or VS Code Debugger to see execution order of asynchronous calls.

Error Handling in Node.js

- Always handle errors gracefully to avoid crashes.
- Use `try...catch` for synchronous code.
- For async callbacks, check the `err` parameter.
- Handle uncaught exceptions centrally.

Example – Synchronous:

```
try {
  const data = JSON.parse('Invalid JSON');
} catch (err) {
  console.error("Error:", err.message);
}
```

Example – Async Callback:

```
const fs = require("fs"); //Use ESM
fs.readFile("nofile.txt", "utf8", (err, data) => {
  if (err) {
    console.error("File not found!");
    return;
  }
  console.log(data);
});
```

Example – Promise:

```
Promise.reject("Something went wrong")
  .catch((err) => console.error("Caught:", err));
```

Global Error Handling:

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

Activity:
Create a JSON parser function that handles invalid input without crashing.

`JSON.parse()` in Node.js (or JavaScript) **throws an error** if the input string is not valid JSON.
Freshers often forget to handle this — and their program **crashes** immediately.

Step 1 – The Problem (Code That Crashes)

```
// crash-demo.js

const invalidJSON = "{ name: 'Alice' }"; // ❌ Invalid JSON (keys must be in quotes)

console.log("Parsing JSON...");

const data = JSON.parse(invalidJSON); // 💥 This line throws SyntaxError

console.log("Parsed successfully:", data);
```

Run:

```
node crash-demo.js
```

Output:

```
Parsing JSON...
node:internal/errors:477
Error [SyntaxError]: Unexpected token n in JSON at position 2
    at JSON.parse (<anonymous>)
    ...
```

Explanation:

- JSON keys and string values **must** be inside double quotes.
- Because there's no error handling, the program **stops immediately**.

Step 2 – The Safe Solution (Using try...catch)

```
// safe-parse-demo.js

function safeParse(jsonString) {
  try {
    const data = JSON.parse(jsonString);
    console.log("✅ JSON parsed successfully!");
    return data;
  } catch (err) {
    console.log("⚠️ Invalid JSON! Error:", err.message);
    return null; // Return null or default object instead of crashing
  }
}

// Test cases
const validJSON = '{"name": "Alice", "age": 25}';
const invalidJSON = "{ name: 'Alice' }";

console.log("\n--- Valid JSON Example ---");
const result1 = safeParse(validJSON);
console.log(result1);

console.log("\n--- Invalid JSON Example ---");
const result2 = safeParse(invalidJSON);
console.log(result2);
```

Run:

```
node safe-parse-demo.js
```

Output:

```
--- Valid JSON Example ---
✅ JSON parsed successfully!
{ name: 'Alice', age: 25 }

--- Invalid JSON Example ---
⚠️ Invalid JSON! Error: Unexpected token n in JSON at position 2
null
```

Working with the Filesystem (CRUD)

Node.js `fs` module enables file operations.

Operation	Method	Example
Create	`fs.writeFile()`	Create new file
Read	`fs.readFile()`	Read contents
Update	`fs.appendFile()`	Add data to file
Delete	`fs.unlink()`	Remove file

Example – CRUD:

```
const fs = require("fs"); // ESM?

// Create
fs.writeFileSync("data.txt", "Hello Node!");

// Read
const data = fs.readFileSync("data.txt", "utf8");
console.log(data);

// Update
fs.appendFileSync("data.txt", "\nNew line added!");

// Delete
fs.unlinkSync("data.txt");
```

Async Version:

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

Tip:
Use `fs.promises` for Promise-based file operations.

Practice Exercise:
Write a script that creates a file `notes.txt`, adds content, reads it, and then deletes it using Promises.

Solution:

```
const fs = require("fs").promises; //ESM?

async function fileOps() {
  await fs.writeFile("notes.txt", "Learning Node.js");
  const data = await fs.readFile("notes.txt", "utf8");
  console.log("File Data:", data);
  await fs.unlink("notes.txt");
}
fileOps();
```

Streams & Buffers Basics

- **Streams** process data chunk by chunk instead of loading it all at once.
- **Buffers** store binary data temporarily.
- Ideal for large files (e.g., video/audio processing).

Types of Streams:

- Readable (e.g., `fs.createReadStream`)
- Writable (e.g., `fs.createWriteStream`)
- Duplex (both read and write)
- Transform (modify data)

Stream Flow:

```
File (input) → Read Stream → Process → Write Stream → Output File
```

Example – Copy File using Streams:

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

readStream.pipe(writeStream);

readStream.on("end", () => console.log("File copied successfully."));
```

Buffer Example:

```
const buf = Buffer.from("Node.js");
console.log(buf.toString());
```

Activity:
Modify the above code to convert file content to uppercase before writing it.

Solution:

```
const fs = require("fs");
const { Transform } = require("stream");

const upperCase = new Transform({
  transform(chunk, enc, cb) {
    cb(null, chunk.toString().toUpperCase());
  },
});

fs.createReadStream("input.txt")
  .pipe(upperCase)
  .pipe(fs.createWriteStream("output.txt"));
```

Building an HTTP Server (Advanced)

Build a simple HTTP server with routes and error handling.

Code:

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

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 if (req.url === "/file") {
    const filePath = path.join(__dirname, "sample.txt");
    fs.readFile(filePath, "utf8", (err, data) => {
      if (err) {
        res.writeHead(404, { "Content-Type": "text/plain" });
        res.end("File Not Found");
        return;
      }
      res.writeHead(200, { "Content-Type": "text/plain" });
      res.end(data);
    });
  } else {
    res.writeHead(404, { "Content-Type": "text/plain" });
    res.end("404 Not Found");
  }
});

server.listen(3000, () => console.log("Server running on
http://localhost:3000"));
```

Tip:
Use Postman or browser to test different routes.

Activity:
Add a `/time` route that returns the current date and time.

Summary

- Node.js modules help organize large codebases.
- Callbacks → Promises → async/await = modern async evolution.
- Always handle errors gracefully to prevent crashes.
- Use Streams for large file processing.
- HTTP module is foundation for building APIs and servers.

Suggested Practice for Participants:

1. Build a mini file management tool (CRUD via CLI).
2. Create a Node.js server that serves JSON data using the `fs` module.
3. Explore the `path` and `os` modules for system information.

---

1) Mini File Management Tool (CRUD via CLI)

Goal: CLI tool to create, read, update (append/overwrite), list, and delete text files in a `notes/` folder.

Save as `file-manager.js`. Run with `node file-manager.js <command> [args]`.

```
// file-manager.js
const fs = require('fs').promises;
const path = require('path');

const NOTES_DIR = path.join(__dirname, 'notes');

async function ensureNotesDir() {
  try {
    await fs.mkdir(NOTES_DIR, { recursive: true });
  } catch (err) {
    console.error('Failed to create notes dir', err);
  }
}

async function createNote(name, content = '') {
  const p = path.join(NOTES_DIR, `${name}.txt`);
  await fs.writeFile(p, content, 'utf8');
  console.log(`Created: ${p}`);
}

async function readNote(name) {
  const p = path.join(NOTES_DIR, `${name}.txt`);
  const data = await fs.readFile(p, 'utf8');
  console.log(`--- ${name} ---\n${data}`);
}

async function appendNote(name, content) {
  const p = path.join(NOTES_DIR, `${name}.txt`);
  await fs.appendFile(p, `\n${content}`, 'utf8');
  console.log(`Appended to: ${p}`);
}

async function overwriteNote(name, content) {
  const p = path.join(NOTES_DIR, `${name}.txt`);
  await fs.writeFile(p, content, 'utf8');
  console.log(`Overwritten: ${p}`);
}

async function deleteNote(name) {
  const p = path.join(NOTES_DIR, `${name}.txt`);
```

```
  await fs.unlink(p);
  console.log(`Deleted: ${p}`);
}

async function listNotes() {
  const files = await fs.readdir(NOTES_DIR);
  const notes = files.filter(f => f.endsWith('.txt')).map(f =>
f.replace(/\.txt$/, ''));
  console.log('Notes:', notes.join(', ') || '(none)');
}

async function main() {
  await ensureNotesDir();
  const [,, cmd, name, ...rest] = process.argv;

  try {
    switch (cmd) {
      case 'create':
        await createNote(name, rest.join(' ') || '');
        break;
      case 'read':
        await readNote(name);
        break;
      case 'append':
        await appendNote(name, rest.join(' '));
        break;
      case 'overwrite':
        await overwriteNote(name, rest.join(' '));
        break;
      case 'delete':
        await deleteNote(name);
        break;
      case 'list':
        await listNotes();
        break;
      default:
        console.log('Usage: node file-manager.js
<create|read|append|overwrite|delete|list> <name> [content]');
    }
  } catch (err) {
    console.error('Error:', err.message);
  }
}

main();
```

Examples:

```
node file-manager.js create meeting "Notes from stand-up"
node file-manager.js append meeting "Added action items"
node file-manager.js read meeting
node file-manager.js list
node file-manager.js delete meeting
```

Tips:

- Show `fs.promises` vs callback API.

- Ask students to add a `rename` command (use `fs.rename`).

---

2) Node.js Server that Serves JSON Data from File

Goal: HTTP server that reads a JSON file and returns JSON for an API endpoint. Use `fs.promises` and simple routing.

Save as `json-server.js` and create `data/users.json`.

`data/users.json`:

```
[
  { "id": 1, "name": "Alice" },
  { "id": 2, "name": "Bob" }
]
```

`json-server.js`:

```
const http = require('http');
const fs = require('fs').promises;
const path = require('path');

const DATA_PATH = path.join(__dirname, 'data', 'users.json');

async function handleRequest(req, res) {
  if (req.url === '/api/users' && req.method === 'GET') {
    try {
      const raw = await fs.readFile(DATA_PATH, 'utf8');
      // Validate that it's valid JSON
      const data = JSON.parse(raw);
      res.writeHead(200, { 'Content-Type': 'application/json' });
      res.end(JSON.stringify({ success: true, data }));
    } catch (err) {
      res.writeHead(500, { 'Content-Type': 'application/json' });
      res.end(JSON.stringify({ success: false, message: 'Failed to read
data' }));
    }
  } else {
    res.writeHead(404, { 'Content-Type': 'application/json' });
    res.end(JSON.stringify({ success: false, message: 'Not found' }));
  }
}

const server = http.createServer((req, res) => {
  // Simple try-catch wrapper: ensures unhandled errors don't crash the
server
  handleRequest(req, res).catch(err => {
    console.error('Unhandled error:', err);
    res.writeHead(500, { 'Content-Type': 'application/json' });
    res.end(JSON.stringify({ success: false, message: 'Server error' }));
  });
});
```

```
server.listen(3000, () => console.log('JSON server listening on
http://localhost:3000'));
```

Run & test:

```
node json-server.js
curl http://localhost:3000/api/users
```

---

3) CLI: Explore `path` and `os` Modules (System Information)

Goal: Small CLI that prints path utilities and OS info.

Save as `sys-info.js` and run `node sys-info.js`.

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

console.log('--- Path utilities ---');
console.log('Current file:', __filename);
console.log('Current directory:', __dirname);
console.log('Join example:', path.join(__dirname, 'data', 'file.txt'));
console.log('Extname example:', path.extname('archive.tar.gz'));
console.log('Basename example:', path.basename('/tmp/some/file.txt'));

console.log('\n--- OS Info ---');
console.log('Platform:', os.platform());          // e.g. 'linux', 'win32'
console.log('Architecture:', os.arch());          // e.g. 'x64'
console.log('CPU count:', os.cpus().length);
console.log('Total memory (MB):', Math.round(os.totalmem()/1024/1024));
console.log('Free memory (MB):', Math.round(os.freemem()/1024/1024));
console.log('Home directory:', os.homedir());
console.log('Uptime (sec):', os.uptime());
```

Exercise ideas:

- Create option flags `node sys-info.js --short` or `--json` to output JSON.
- Show how to normalize paths across OSes (`path.normalize`).

---

4) Stream-based File Copy + Transform (Uppercase)

Goal: Demonstrate streaming, transform streams, backpressure and memory efficiency.

Save as `stream-uppercase.js`.

```
const fs = require('fs');
const { Transform } = require('stream');
const path = require('path');

const input = path.join(__dirname, 'large-input.txt');
```

```
const output = path.join(__dirname, 'large-output.txt');

const upperTransform = new Transform({
  transform(chunk, encoding, callback) {
    // chunk may be Buffer — convert to string, uppercase, back to Buffer
    const result = chunk.toString().toUpperCase();
    callback(null, Buffer.from(result));
  }
});

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

readStream
  .pipe(upperTransform)
  .pipe(writeStream)
  .on('finish', () => console.log('Streaming transform complete'));
```

To test: create `large-input.txt` (copy/paste, or `yes "some data" | head -n 10000 > large-input.txt`) and run:

```
node stream-uppercase.js
```

Discussion points:

- Memory usage vs `fs.readFile` for large files.
- What is backpressure and how `pipe()` handles it.

---

5) File Operations with Promises — Notes App Mini-API (CLI + JSON DB)

Goal: Use `fs.promises` to implement a JSON-based note storage with simple commands.

Save as `notes-api.js`. Usage examples follow.

```
// notes-api.js
const fs = require('fs').promises;
const path = require('path');

const DB = path.join(__dirname, 'notes-db.json');

async function loadDB() {
  try {
    const txt = await fs.readFile(DB, 'utf8');
    return JSON.parse(txt || '[]');
  } catch (err) {
    // If file doesn't exist, return empty array
    if (err.code === 'ENOENT') return [];
    throw err;
  }
}
```

```
async function saveDB(data) {
  await fs.writeFile(DB, JSON.stringify(data, null, 2), 'utf8');
}

async function addNote(title, body) {
  const db = await loadDB();
  const id = db.length ? db[db.length - 1].id + 1 : 1;
  db.push({ id, title, body, createdAt: new Date().toISOString() });
  await saveDB(db);
  console.log('Added note id', id);
}

async function listNotes() {
  const db = await loadDB();
  console.table(db.map(({id, title, createdAt}) => ({id, title,
createdAt})));
}

async function getNote(id) {
  const db = await loadDB();
  const note = db.find(n => n.id === Number(id));
  if (!note) console.log('Not found');
  else console.log(note);
}

async function deleteNote(id) {
  let db = await loadDB();
  const before = db.length;
  db = db.filter(n => n.id !== Number(id));
  await saveDB(db);
  console.log(before === db.length ? 'Not found' : 'Deleted');
}

// CLI dispatch
(async function(){
  const [,, cmd, arg1, ...rest] = process.argv;
  try {
    switch(cmd) {
      case 'add':
        await addNote(arg1 || 'Untitled', rest.join(' ') || '');
        break;
      case 'list':
        await listNotes();
        break;
      case 'get':
        await getNote(arg1);
        break;
      case 'delete':
        await deleteNote(arg1);
        break;
      default:
        console.log('Usage: node notes-api.js <add|list|get|delete>
[args]');
    }
  } catch (err) {
    console.error('Error:', err.message);
  }
})();
```

Examples:

```
node notes-api.js add "Todo" "Finish Node.js workshop"
node notes-api.js list
node notes-api.js get 1
node notes-api.js delete 1
```

Extension: Add `update` command or add validation to prevent duplicate titles.

---

6) Error Handling Patterns — Demo

Goal: Short code snippet to show handling sync, async, and unhandled errors.

```
// error-demo.js
const fs = require('fs').promises;

async function run() {
  try {
    // synchronous error
    JSON.parse('not valid json'); // will throw in try/catch
  } catch (err) {
    console.error('Caught sync error:', err.message);
  }

  try {
    await fs.readFile('no-such-file.txt','utf8'); // will reject
  } catch (err) {
    console.error('Caught async error:', err.code);
  }

  // simulate unhandled rejection
  Promise.reject(new Error('unhandled rejection example'));
}

process.on('unhandledRejection', (reason, p) => {
  console.error('Unhandled Rejection at:', p, 'reason:', reason.message);
});

process.on('uncaughtException', (err) => {
  console.error('Uncaught Exception:', err.message);
  // In real servers, consider graceful shutdown after logging
});

run();
```

note: relying only on global handlers is not a substitute for proper per-operation error handling.

---

7) Exercise — Implement `/time` route

Goal: Extend the earlier HTTP server to include `GET /time` to return current time.

```
// time-server.js (extends basic routing)
const http = require('http');

const server = http.createServer((req, res) => {
  if (req.url === '/time' && req.method === 'GET') {
    res.writeHead(200, {'Content-Type': 'application/json'});
    res.end(JSON.stringify({ now: new Date().toISOString() }));
    return;
  }

  // other routes...
  res.writeHead(404, {'Content-Type': 'application/json'});
  res.end(JSON.stringify({ error: 'Not found' }));
});

server.listen(3001, () => console.log('Time server on
http://localhost:3001'));
```

Test:

```
curl http://localhost:3001/time
# => {"now":"2025-10-24T10:12:34.000Z"}
```