NODE.JS

Fundamentals

Event Loop · Modules & npm · Callbacks & Promises

Async/Await · File System · HTTP Servers

Debugging · Best Practices

Philip Osztromok

Generated with Claude

Table of Contents

Node.js Fundamentals — 8 Chapters

⚡ What is Node.js & the Event Loop

Node.js is a JavaScript runtime that lets you run JavaScript outside the browser — typically on servers and command-line tools. The magic behind Node.js is the event loop: a non-blocking, asynchronous execution model that makes it incredibly efficient at handling I/O operations. This chapter explains what Node.js is, why the event loop matters, and how asynchronous programming works.

🌐 What is Node.js?

Node.js is:

- **A JavaScript runtime** — runs JavaScript code outside the browser
- **Built on Chrome's V8 engine** — the same high-performance JavaScript engine that powers Google Chrome
- **Event-driven** — built around an event loop for handling asynchronous operations
- **Non-blocking I/O** — performs file/network operations without blocking execution
- **Single-threaded** — your JavaScript runs on a single thread (though Node uses threads behind the scenes for I/O)

Browser JavaScript vs Node.js

Browser (Client-Side)

- Runs in the user's browser
- Accesses DOM, local storage
- Handles user interactions
- Limited file system access
- Same-origin policy restrictions

Node.js (Server-Side)

- Runs on servers/your machine
- Full file system access
- No DOM (no UI)
- Database/network operations
- Environment variables, secrets

🔄 The Event Loop: The Heart of Node.js

The event loop is Node.js's execution model. It allows Node.js to handle thousands of concurrent connections without creating a thread for each one.

```
// This code demonstrates the event loop in action

console.log('1. Start');

setTimeout(() => {
  console.log('2. Timeout callback (async)');
}, 0);

console.log('3. End');

// Output:
// 1. Start
// 3. End
// 2. Timeout callback (async)
//
// Why? The setTimeout is async — it goes to the event loop queue
// and the main code continues. After the call stack is empty,
// the event loop processes the queued callback.
```

Call Stack

Where your synchronous code executes. Functions are pushed and popped as they run.

Event Loop

Checks if the call stack is empty, then processes callbacks from the queue.

Callback Queue

Where async callbacks (setTimeout, file reads, etc.) wait to be executed.

Non-Blocking

I/O operations don't pause execution — they're handled asynchronously.

⛔ Blocking vs Non-Blocking I/O

Two Approaches to File Reading

Blocking (❌ Don't do this)

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

// This BLOCKS execution
const data = fs.readFileSync('file.txt', 'utf8');
console.log(data);  // Waits for file to load
// Other code is stuck waiting
console.log('This runs after file loads');
```

Non-Blocking (✅ Do this)

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

// This DOESN'T block execution
fs.readFile('file.txt', 'utf8', (err, data) => {
  console.log(data);  // Callback when file loads
});

// Other code runs immediately
console.log('This runs right away');
```

📞 Callbacks: The Foundation of Async

A callback is a function passed to another function, to be called later when something happens:

```
function processFile(filename, callback) {
  // Read file asynchronously
  fs.readFile(filename, 'utf8', (err, data) => {
    if (err) {
      callback(err, null);  // Error callback
    } else {
      callback(null, data);   // Success callback
    }
  });
}

// Usage: pass a callback
processFile('data.txt', (err, data) => {
  if (err) console.error('Error:', err);
  else console.log('Data:', data);
});
```

Note: Modern Node.js also supports Promises and async/await (which we'll cover in later chapters). But callbacks are fundamental to understanding how Node.js works.

▶️ Running Node.js

Your First Node.js Program

```
// hello.js
console.log('Hello, Node.js!');
console.log('Node version:', process.version);
```

Run it:

```
$ node hello.js
Hello, Node.js!
Node version: v18.0.0
```

Interactive REPL: Type `node` to enter the interactive shell:

```
$ node
> console.log('Hello!')
Hello!
> 2 + 2
4
> .exit
```

💻 Coding Challenges

Challenge 1: Understanding Execution Order

Write a Node.js script that logs numbers in different orders using synchronous code, setTimeout, and callbacks. Predict the output before running it.

Goal: Internalize how the event loop affects execution order.

→ Solution

Challenge 2: Blocking vs Non-Blocking

Create two versions of a program that reads a file: one using readFileSync (blocking) and one using readFile (non-blocking). Measure which is faster when reading multiple files.

Goal: See non-blocking I/O in action.

→ Solution

Challenge 3: Callback Pattern

Write a function that accepts a callback and calls it with data from a file. Use error-first callbacks (error as first param). Test both success and error paths.

Goal: Master the callback pattern used throughout Node.js.

→ Solution

💡 Node.js Philosophy: "No I/O blocking"

The golden rule of Node.js is: never block the event loop. Even a 1-second file read blocks everything. This is why non-blocking I/O is so important. Later chapters cover Promises and async/await, which make non-blocking code easier to read.

🎯 What's Next

Now that you understand what Node.js is and how the event loop works, we'll dive into **Modules & npm** — how to organize code and manage dependencies.

📦 Modules & npm/package.json

Node.js applications are built from modules — reusable pieces of code. The module system lets you organize code, avoid global scope pollution, and share code between files. npm (Node Package Manager) lets you download and manage packages (libraries) that others have published. The package.json file is the heart of dependency management. This chapter covers how to create and use modules, and how to manage packages with npm.

🧩 The Module System: CommonJS

Node.js uses **CommonJS** modules. A module is just a JavaScript file that exports code for other files to use.

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

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

// Export the functions
module.exports = {
  add,
  subtract
};


// app.js — Use the module
const math = require('./math');

console.log(math.add(5, 3));      // 8
console.log(math.subtract(5, 3)); // 2
```

module.exports

What a module shares. Can be a function, object, class, or anything else.

require()

Load a module. Returns what the module exported.

Local Modules

`require('./path')` — load a file in your project.

Installed Packages

`require('lodash')` — load an npm package.

📥 npm: Package Manager

npm is Node's package manager. It lets you:

- Download and install packages (libraries) that others have written
- Manage versions of packages your project depends on
- Publish your own packages for others to use
- Run scripts defined in package.json

Installing a Package

```
$ npm install lodash
This:
1. Downloads lodash from npmjs.com
2. Stores it in node_modules/ folder
3. Updates package.json and package-lock.json
Then use it:
const _ = require('lodash');
console.log(_.shuffle([1, 2, 3, 4, 5]));
```

📄 package.json: Your Project's Config

package.json is a JSON file that describes your project and its dependencies.

```
{
 "name": "my-awesome-app",
 "version": "1.0.0",
 "description": "My first Node.js app",
 "main": "app.js",
 "scripts": {
  "start": "node app.js",
  "test": "jest",
  "dev": "nodemon app.js"
 },
 "dependencies": {
  "express": "^4.18.0",
  "lodash": "^4.17.21"
 },
 "devDependencies": {
  "nodemon": "^2.0.20",
  "jest": "^29.0.0"
 },
 "author": "Your Name",
 "license": "MIT"
}
```

name & version

Package identity. Follows semantic versioning (1.0.0 = major.minor.patch).

scripts

Commands you can run: `npm start`, `npm test`, `npm run dev`.

dependencies

Packages your app needs to run (installed in production).

devDependencies

Packages only needed for development (testing, linting, bundling).

🆕 Creating package.json

Interactive Setup

```
$ npm init
Prompts for: name, version, description, entry point, test command, etc.
Creates package.json with your answers.
```

Quick Setup (use defaults)

```
$ npm init -y
Creates package.json with default values (faster).
```

🔢 Semantic Versioning (Semver)

Versions follow the pattern `MAJOR.MINOR.PATCH` (e.g., 1.5.3):

- **1**.0.0 — Breaking changes (incompatible API)
- 1.**5**.0 — New features (backward compatible)
- 1.5.**3** — Bug fixes (backward compatible)

Version Constraints in package.json

Exact Version

```
"lodash": "4.17.21"
// Installs exactly 4.17.21
// No updates, even if 4.17.22 exists
```

Flexible Version

```
"lodash": "^4.17.21"
// Allows 4.17.21 to 4.99.99
// ^ = compatible minor/patch
// ~ = only patch updates
```

💻 Coding Challenges

Challenge 1: Create & Use Modules

Create a `utils.js` module that exports functions for validation (email, phone number). Create `app.js` that requires and uses these functions.

Goal: Practice module creation and require().

→ Solution

Challenge 2: npm & package.json

Initialize a new Node.js project with npm init. Install lodash and express. Add a custom npm script that echoes "Hello from npm script". Run it.

Goal: Get comfortable with npm and package.json.

→ Solution

Challenge 3: Exporting Different Types

Create modules that export: a function, an object, a class, and a constant. From app.js, require each and use them in different ways.

Goal: Understand that module.exports can be anything.

→ Solution

💡 node_modules/ is Big

After you npm install packages, a huge `node_modules/` folder is created. Don't commit this to git — add it to .gitignore. When cloning a repo, just run `npm install` and it recreates node_modules from package.json. This keeps your repo small and reproducible.

🎯 What's Next

Now that you can organize code with modules and manage packages with npm, we'll explore **Callbacks & Promises** — diving deeper into async patterns beyond setTimeout.

🔄 Callbacks & Promises

Asynchronous programming is core to Node.js. We've seen callbacks briefly; now we dive deeper into why they matter and their problems. We'll also introduce **Promises** — a cleaner way to handle async operations and avoid "callback hell." This chapter covers callback patterns, promise basics, and promise chains.

📞 Callbacks Revisited

A callback is a function passed to another function, called when an operation completes:

```
function fetchUser(id, callback) {
  // Simulate async operation (like database query)
setTimeout(() => {
    const user = { id, name: 'Alice' };
    callback(null, user);
  }, 1000);
}

// Use it
fetchUser(1, (err, user) => {
  if (err) {
    console.error('Error:', err);
  } else {
    console.log('User:', user);
  }
});
```

⚠️ Callback Hell (Pyramid of Doom)

When callbacks nest deeply, code becomes hard to read and maintain:

The Problem

```
getUser(1, (err, user) => {
  if (err) {
    console.error(err);
  } else {
    getOrders(user.id, (err, orders) => {
      if (err) {
        console.error(err);
      } else {
        getOrderDetails(orders[0].id, (err, details) => {
          if (err) {
            console.error(err);
          } else {
            console.log('Order details:', details);
          }
        });
      }
    });
  }
});
```

Each nested operation adds another level of indentation. This is hard to debug and modify.

✨ Promises: A Better Way

A **Promise** represents a value that will be available in the future. It has three states:

Pending

Operation hasn't finished yet. Waiting for completion.

Resolved (Fulfilled)

Operation succeeded. Promise has a value.

Rejected

Operation failed. Promise has an error.

Settled

Either resolved or rejected. No longer pending.

Creating a Promise

```
const promise = new Promise((resolve, reject) => {
  // Async operation
setTimeout(() => {
    const success = true;
    if (success) {
      resolve('Operation succeeded!'); // Fulfilled
    } else {
      reject(new Error('Operation failed!')); // Rejected
    }
  }, 1000);
});

promise
  .then(result => console.log(result))       // Runs if resolved
  .catch(error => console.error(error));  // Runs if rejected
```

🔗 .then() and .catch()

`.then()` runs when the promise resolves. `.catch()` runs when it rejects.

```
function fetchData(id) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (id > 0) {
        resolve({ id, data: 'Some data' });
      } else {
        reject(new Error('Invalid ID'));
      }
    }, 1000);
  });
}

// Success case
fetchData(1)
  .then(result => console.log('Got:', result))
  .catch(error => console.error('Error:', error.message));

// Error case
fetchData(-1)
  .then(result => console.log('Got:', result))
  .catch(error => console.error('Error:', error.message));
```

⛓️ Promise Chains

Chain multiple operations. Each `.then()` passes its result to the next:

Callback Hell vs Promise Chains

Callbacks (Hard to Read)

```
getUser(1, (err, user) => {
  if (err) {
    console.error(err);
  } else {
    getOrders(user.id, (err, orders) => {
      if (err) {
        console.error(err);
      } else {
        console.log(orders);
      }
    });
  }
});
```

Promises (Readable)

```
getUser(1)
  .then(user => getOrders(user.id))
  .then(orders => console.log(orders))
  .catch(error => console.error(error));
```

How Promise Chains Work

```
fetchUser(1)
 .then(user => {
  console.log('User:', user);
  return fetchOrders(user.id); // Returns a promise
 })
 .then(orders => {
  console.log('Orders:', orders);
  return orders[0].id; // Return a value (auto-wrapped in promise)
 })
 .then(orderId => {
  console.log('First order ID:', orderId);
 })
 .catch(error => {
  console.error('Error in chain:', error);
 });
```

Key point: Each `.then()` receives the resolved value from the previous `.then()`. If you return a promise, the next `.then()` waits for it. If you return a value, it's automatically wrapped in a resolved promise.

🔀 Parallel Operations: Promise.all()

Run multiple promises in parallel and wait for all to complete:

```
const p1 = fetchUser(1);
const p2 = fetchUser(2);
const p3 = fetchUser(3);

Promise.all([p1, p2, p3])
 .then(([user1, user2, user3]) => {
   console.log('All users:', user1, user2, user3);
 })
 .catch(error => {
   console.error('One of them failed:', error);
 });
```

Note: If any promise rejects, `Promise.all()` rejects immediately.

⚡ First to Finish: Promise.race()

Waits for the first promise to settle (resolve or reject):

```
const timeout = new Promise((_, reject) =>
  setTimeout(() => reject(new Error('Timeout!')), 5000)
);

Promise.race([fetchData(), timeout])
  .then(result => console.log('Got result:', result))
  .catch(error => console.error('Error or timeout:', error.message));
```

💻 Coding Challenges

Challenge 1: Convert Callbacks to Promises

Take a callback-based function and convert it to return a promise. Then chain multiple calls using `.then()`.

Goal: Practice promise creation and chaining.

→ Solution

Challenge 2: Promise.all() for Parallel Operations

Create three promise-returning functions that fetch different data. Use `Promise.all()` to fetch all three in parallel and log the combined results.

Goal: Understand parallel promise execution.

→ Solution

Challenge 3: Error Handling in Chains

Create a promise chain where one operation fails. Show how the error propagates down to `.catch()` and how to recover from errors mid-chain.

Goal: Master promise error handling.

→ Solution

💡 Promise vs Callback Cheat Sheet

Callbacks: Require error checking in every callback. Nest deeply for sequential operations. Hard to debug.

Promises: Error handling with a single `.catch()`. Cleaner chains. Better error propagation.

Next: async/await makes promises even cleaner — but understanding promises is the foundation!

🎯 What's Next

Promises are powerful, but we can make them even easier to read with **async/await** — JavaScript's modern async syntax. We'll explore that in the next chapter.

⏳ Async/Await

`async/await` is syntactic sugar built on top of promises. It lets you write asynchronous code that looks and feels like synchronous code — no more chains of `.then()` calls. This makes async code cleaner, easier to read, and easier to debug. This chapter covers async functions, await expressions, error handling with try/catch, and concurrent operations with async/await.

🔧 Async Functions

An `async` function always returns a promise:

```
async function greet() {
  return 'Hello!';
}

// This is equivalent to:
function greet() {
  return Promise.resolve('Hello!');
}

// Both return a promise
greet().then(msg => console.log(msg)); // Hello!
```

The value you return from an async function is automatically wrapped in a resolved promise.

⏸️ The await Keyword

`await` pauses execution until a promise settles. It can only be used inside an async function:

```
async function fetchUser(id) {
  // Pause here until promise resolves
const user = await getUserFromDB(id);
  console.log('User:', user);
  return user;
}

// Call the async function
const result = fetchUser(1); // Returns a promise
result.then(user => console.log(user));
```

📊 Async/Await vs Promises

Side-by-Side Comparison

Promise Chains

```
fetchUser(1)
 .then(user => {
  console.log(user);
  return fetchOrders(user.id);
 })
 .then(orders => {
  console.log(orders);
 })
 .catch(error => {
  console.error(error);
 });
```

Async/Await

```
async function load() {
 try {
  const user = await fetchUser(1);
  console.log(user);
  const orders = await fetchOrders(user.id);
  console.log(orders);
 } catch (error) {
  console.error(error);
 }
}

load();
```

Async/await reads like synchronous code, making it much easier to follow.

🛡️ Error Handling with Try/Catch

Use `try`/`catch` to handle errors in async functions:

```
async function fetchUserData(id) {
  try {
    const user = await fetchUser(id);
    const orders = await fetchOrders(user.id);
    return { user, orders };
  } catch (error) {
    console.error('Failed to load data:', error.message);
    return null; // Return fallback value
  }
}

// Call it
await fetchUserData(1);
```

If any `await` expression rejects, execution jumps to `catch`.

⚡ Sequential vs Concurrent Execution

Sequential (Slow)

```
async function sequential() {
  const user = await fetchUser(1);      // 1000ms
const posts = await fetchPosts(user.id); // 1000ms
return { user, posts };               // Total: 2000ms
}
```

Concurrent (Fast)

```
async function concurrent() {
  // Start both promises, then await results
const userPromise = fetchUser(1);     // Start here
const postsPromise = fetchPosts(1);   // Start here
// Wait for both
const user = await userPromise;
  const posts = await postsPromise;
  return { user, posts };               // Total: ~1000ms
}

// Or use Promise.all():
async function concurrent() {
  const [user, posts] = await Promise.all([
    fetchUser(1),
    fetchPosts(1)
  ]);
  return { user, posts };
}
```

🎯 Arrow Async Functions

```
const fetchAndLog = async (id) => {
  const user = await fetchUser(id);
  console.log(user);
};

// Usage
await fetchAndLog(1);
```

🚀 Top-Level Await (Modern Node.js)

In recent Node.js versions, you can use `await` at the top level of a module (no async function needed):

```
// Old way: wrap in async function
async function main() {
  const data = await fetchData();
  console.log(data);
}
main();

// New way: use await directly
const data = await fetchData();
console.log(data);
```

💻 Coding Challenges

Challenge 1: Convert Promise Chain to Async/Await

Take a promise chain and rewrite it using async/await. Compare readability.

Goal: Practice converting between async patterns.

→ Solution

Challenge 2: Sequential vs Concurrent Operations

Create an async function that fetches three datasets. Implement both sequential and concurrent versions. Measure the time difference.

Goal: Understand performance implications of await placement.

→ Solution

Challenge 3: Error Handling with Try/Catch

Create an async function with multiple awaits and error handling. Show recovery, logging, and finally blocks.

Goal: Master error handling in async code.

→ Solution

💡 Async/Await Best Practices

1. Don't forget await: Forgetting `await` returns a promise instead of the value.

2. Use concurrent execution: Start independent operations before awaiting them.

3. Always handle errors: Use try/catch or .catch() on the returned promise.

4. Avoid await in loops if they're independent: Use Promise.all() instead.

🎯 What's Next

Now that you can handle asynchronous code cleanly with async/await, we'll explore **File System Operations** — reading, writing, and managing files in Node.js using these async patterns.

📁 File System Operations

Node.js provides the `fs` (File System) module to interact with files and directories. You can read, write, delete, and manipulate files. This chapter covers reading files, writing files, working with directories, using async/await with the promises API, and common patterns for file operations.

📖 The fs Module

The `fs` module is built into Node.js. Import it to access file operations:

```
const fs = require('fs');
const { promises: fsp } = require('fs');

// Or in modern Node.js:
import { promises as fs } from 'fs';
```

The fs module provides three ways to work with files:

Synchronous (Blocking)

`readFileSync()`, `writeFileSync()` — blocks execution. Avoid in production!

Callback (Old)

`readFile()`, `writeFile()` with callback — harder to use.

Promises (Modern)

`fs.promises.readFile()` — returns promise, works with async/await.

Streams

For large files — efficient memory usage (we'll cover later).

📚 Reading Files

Read Text File (Async/Await)

```
const { promises: fs } = require('fs');

async function readTextFile(filename) {
  try {
    const content = await fs.readFile(filename, 'utf8');
    console.log('File content:', content);
    return content;
  } catch (error) {
    console.error('Error reading file:', error.message);
  }
}

await readTextFile('data.txt');
```

Read as Buffer (Binary Data)

```
const data = await fs.readFile('image.png');
console.log('Buffer length:', data.length); // bytes
```

✍️ Writing Files

Write Text (Creates or Overwrites)

```
async function writeFile(filename, content) {
  try {
    await fs.writeFile(filename, content, 'utf8');
    console.log('✓ File written');
  } catch (error) {
    console.error('Error writing file:', error.message);
  }
}

await writeFile('output.txt', 'Hello, World!');
```

Append to File

```
async function appendFile(filename, content) {
  await fs.appendFile(filename, content, 'utf8');
  console.log('✓ Content appended');
}

await appendFile('log.txt', '\nNew log entry');
```

📂 Directory Operations

List Files in Directory

```
async function listFiles(dirname) {
 try {
  const files = await fs.readdir(dirname);
  console.log('Files:', files);
  return files;
 } catch (error) {
  console.error('Error reading directory:', error.message);
 }
}

await listFiles('./data');
```

Create Directory

```
async function createDir(dirname) {
 await fs.mkdir(dirname, { recursive: true });
 console.log('✓ Directory created');
}

await createDir('./output/subdir');
```

ℹ️ File Information

```
async function getFileStats(filename) {
  const stats = await fs.stat(filename);

  console.log('File size:', stats.size, 'bytes');
  console.log('Is file?', stats.isFile());
  console.log('Is directory?', stats.isDirectory());
  console.log('Modified:', stats.mtime);
}

await getFileStats('data.txt');
```

🗑️ Deleting Files & Directories

```
// Delete a file
await fs.unlink('old-file.txt');

// Delete a directory (must be empty)
await fs.rmdir('./empty-dir');

// Delete recursively (Node.js 14.14+)
await fs.rm('./dir-with-files', { recursive: true });
```

🔄 Common Patterns

Process JSON File

```
async function loadConfig(filename) {
  const json = await fs.readFile(filename, 'utf8');
  const config = JSON.parse(json);
  return config;
}

const config = await loadConfig('config.json');
```

Copy File

```
async function copyFile(src, dest) {
  const content = await fs.readFile(src);
  await fs.writeFile(dest, content);
  console.log('✓ File copied');
}

await copyFile('original.txt', 'backup.txt');
```

💻 Coding Challenges

Challenge 1: Read & Process Text File

Read a text file, count lines/words, and write statistics to a new file.

Goal: Practice reading files and basic text processing.

→ Solution

Challenge 2: JSON Config Management

Read a JSON config file, modify values, and write it back. Handle missing files gracefully.

Goal: Work with JSON files and error handling.

→ Solution

Challenge 3: Directory Operations

List all files in a directory, filter by extension, and generate a file report.

Goal: Master directory reading and file metadata.

→ Solution

💡 File I/O Best Practices

1. Always use async/await: Never use synchronous file operations in production code.

2. Handle errors: Files might not exist, disk might be full, permissions might be denied.

3. Use streams for large files: Reading a 10GB file into memory is a bad idea!

4. Consider paths carefully: Use `path.join()` for cross-platform compatibility.

🎯 What's Next

Now that you can read and write files, we'll explore **HTTP Servers** — using Node.js to create web servers and handle HTTP requests.

🌐 HTTP Servers

The `http` module lets you create web servers in Node.js. Clients send HTTP requests to your server, and your server sends back HTTP responses. This chapter covers creating a basic HTTP server, handling requests, sending responses, parsing URLs, and routing requests to different handlers.

📡 HTTP Basics

HTTP is a request-response protocol:

Request

Client sends: method, URL, headers, body.

Response

Server sends: status code, headers, body.

Methods

GET (read), POST (create), PUT (update), DELETE (remove).

Status Codes

200 (OK), 404 (Not Found), 500 (Error), etc.

Creating a Basic Server

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

const server = http.createServer((req, res) => {
  // req: the request object
// res: the response object

  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, World!');
});

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

Run with `node server.js`, then visit `http://localhost:3000` in your browser.

📥 The Request Object

The `req` object contains information about the client's request:

```
const server = http.createServer((req, res) => {
  console.log('Method:', req.method);     // GET, POST, etc.
  console.log('URL:', req.url);           // /path?query=value
  console.log('Headers:', req.headers);   // {host, user-agent, ...}

  res.end('OK');
});
```

📤 The Response Object

Use the `res` object to send a response:

```
// Set status code
res.statusCode = 200;

// Set headers
res.setHeader('Content-Type', 'text/html');
res.setHeader('X-Custom-Header', 'value');

// Write body
res.write('
<h1>Hello</h1>
');
res.write('
<p>World</p>
');

// End response (required!)
res.end();
```

🛣️ Routing Requests

Handle different URLs differently:

```
const server = http.createServer((req, res) => {
  if (req.url === '/' && req.method === 'GET') {
    res.statusCode = 200;
    res.end('Home Page');
  } else if (req.url === '/about') {
    res.statusCode = 200;
    res.end('About Page');
  } else {
    res.statusCode = 404;
    res.end('Not Found');
  }
});
```

🔍 Parsing Requests

Parse URL and Query Parameters

```
const { URL } = require('url');

const server = http.createServer((req, res) => {
  const url = new URL(req.url, `http://${req.headers.host}`);

  console.log('Pathname:', url.pathname);    // /search
  console.log('Query:', url.search);      // ?q=node
const query = url.searchParams.get('q');
  console.log('Search query:', query);       // node

  res.end('OK');
});
```

📋 JSON Responses

```
const server = http.createServer((req, res) => {
  const data = {
    message: 'Hello, World!',
    timestamp: new Date()
  };

  res.setHeader('Content-Type', 'application/json');
  res.end(JSON.stringify(data));
});
```

Handling POST Requests

```
const server = http.createServer(async (req, res) => {
  if (req.method === 'POST') {
    let body = '';

    // Collect request body
    req.on('data', chunk => {
      body += chunk.toString();
    });

    // Process when finished
    req.on('end', () => {
      const data = JSON.parse(body);
      console.log('Received:', data);

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

📂 Serving Static Files

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

const server = http.createServer(async (req, res) => {
 const filePath = path.join(__dirname, 'public', req.url);

 try {
  const data = await fs.promises.readFile(filePath);
  res.statusCode = 200;
  res.end(data);
 } catch (error) {
  res.statusCode = 404;
  res.end('Not Found');
 }
});
```

🔢 Common Status Codes

2xx Success

200 OK, 201 Created, 204 No Content

3xx Redirect

301 Moved, 302 Found, 304 Not Modified

4xx Client Error

400 Bad Request, 401 Unauthorized, 404 Not Found

5xx Server Error

500 Internal Error, 502 Bad Gateway, 503 Unavailable

💻 Coding Challenges

Challenge 1: Basic HTTP Server

Create a server that responds to /, /about, /api/users routes with different responses.

Goal: Learn server creation and basic routing.

→ Solution

Challenge 2: JSON API

Create a server that accepts GET/POST requests and responds with JSON data.

Goal: Handle requests and build a simple API.

→ Solution

Challenge 3: Serve Static Files

Create a server that serves HTML, CSS, and image files from a public directory.

Goal: Understand content types and static file serving.

→ Solution

💡 Server Best Practices

1. Always end responses: Call `res.end()` or data won't be sent.

2. Set correct content types: HTML, JSON, images need different MIME types.

3. Handle errors: Files might not exist, requests might be malformed.

4. Use Express later: For production, use frameworks like Express.js (we'll cover in exp1).

🎯 What's Next

You've created HTTP servers with Node.js! We'll explore more advanced topics, and soon move to **Express.js** — a powerful web framework that makes building servers much easier.

🐛 Debugging & Error Handling

Errors are inevitable in programming. This chapter teaches you to identify, handle, and debug errors effectively. We'll cover error types, try/catch patterns, logging strategies, debugging tools, and best practices for robust error handling.

📋 Error Types

JavaScript has built-in error types:

Error

Base error class. Generic errors.

TypeError

Wrong type (e.g., calling non-function).

ReferenceError

Undefined variable or function.

SyntaxError

Invalid syntax (caught at parse time).

RangeError

Value out of valid range.

Custom Errors

Create your own error classes.

📝 Console Debugging

```
// Basic logging
console.log('Info message');
console.warn('Warning');
console.error('Error message');

// Debug levels
console.debug('Debug info');
console.info('Info');

// Structured output
console.table(data);
console.dir(obj);

// Timing
console.time('myTimer');
// ... code ...
console.timeEnd('myTimer');

// Assertions
console.assert(value > 0, 'Value must be positive');
```

🎯 Try/Catch Blocks

```
try {
  // Code that might throw
const result = riskyOperation();
} catch (error) {
  // Handle specific errors
if (error instanceof TypeError) {
    console.error('Type error:', error.message);
  } else {
    console.error('Unknown error:', error);
  }
} finally {
  // Cleanup (always runs)
  cleanup();
}
```

⚙️ Custom Error Classes

```
class ValidationError extends Error {
  constructor(message, field) {
    super(message);
    this.name = 'ValidationError';
    this.field = field;
  }
}

throw new ValidationError('Invalid email', 'email');
```

📊 Logging Strategies

Log Levels

```
// DEBUG: Detailed debugging info
console.debug('User ID: 123');

// INFO: General information
console.log('Server started');

// WARN: Warning (recoverable)
console.warn('High memory usage');

// ERROR: Error (needs attention)
console.error('Database connection failed');
```

🔄 Async Error Handling

Promise Errors

```
async function loadData() {
  try {
    const data = await fetchData();
    return data;
  } catch (error) {
    console.error('Failed to load:', error.message);
    throw error;
  }
}

// OR with .catch()
fetchData()
  .catch(error => {
    console.error('Error:', error);
  });
```

📤 Error Propagation

Handle or Propagate?

Don't Swallow Errors

```
try {
  riskyOp();
} catch (e) {
  // Empty! Bad!
}
```

Log or Re-throw

```
try {
  riskyOp();
} catch (e) {
  console.error(e);
  throw e;
}
```

🔍 Stack Traces

A stack trace shows the call chain when an error occurs:

```
TypeError: Cannot read property 'name' of undefined
  at getUser (/app/users.js:5:10)
  at getUserName (/app/app.js:12:5)
  at main (/app/app.js:20:3)

// Read from bottom to top (main → getUserName → getUser)
// Error happened in getUser at line 5, column 10
```

🛠️ Debugging Tools

Node.js Debugger

```
$ node inspect app.js
$ node --inspect app.js
Then open Chrome DevTools:
chrome://inspect
```

💻 Coding Challenges

Challenge 1: Custom Error Classes

Create custom error classes (ValidationError, DatabaseError) and use them in functions.

Goal: Learn to create and throw meaningful errors.

→ Solution

Challenge 2: Async Error Handling

Create async functions with proper try/catch and error propagation patterns.

Goal: Master async error handling in promises and async/await.

→ Solution

Challenge 3: Logging System

Build a simple logger with levels (debug, info, warn, error) and timestamps.

Goal: Understand structured logging for production apps.

→ Solution

💡 Error Handling Best Practices

1. Never swallow errors silently: Always log or re-throw.

2. Be specific: Create custom error classes for different scenarios.

3. Provide context: Include useful information in error messages.

4. Use proper status codes: 400 for bad requests, 500 for server errors.

5. Log at startup: Output config and status on app start.

🎯 What's Next

You've learned debugging and error handling! The Node.js Fundamentals course is nearly complete. We'll wrap up with **Best Practices & Wrap-Up** in the final chapter (node1-8).

🎓 Best Practices & Wrap-Up

You've learned the fundamentals of Node.js! This final chapter reinforces best practices, covers security basics, performance optimization, and recaps everything you've learned. We'll also discuss next steps and resources for continued learning.

✨ Code Quality

Write Clean Code

- Use meaningful variable names
- Keep functions small and focused
- Avoid deep nesting (max 3 levels)
- DRY: Don't Repeat Yourself
- Use consistent formatting and style

🔄 Async Best Practices

Async/Await is Better

Callbacks (Avoid)

```
getData(function(err, data) {
  if (err) {
    handleError(err);
  } else {
    processData(data);
  }
});
```

Async/Await (Use This)

```
try {
  const data = await getData();
  processData(data);
} catch (err) {
  handleError(err);
}
```

🛡️ Error Handling

Always Handle Errors

- Use try/catch in async functions
- Never swallow errors silently
- Log errors with context
- Use custom error classes
- Return appropriate status codes
- Don't expose sensitive info in errors

🔐 Security Basics

Prevent Command Injection

```
❌ DANGEROUS
const cmd = `ls ${userInput}`;
const result = execSync(cmd);

✅ SAFE
const result = execFile('ls', [userInput]);
```

Validate Input

```
✅ SAFE
function validateEmail(email) {
 const regex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
 if (!regex.test(email)) {
  throw new ValidationError('Invalid email');
 }
}
```

Use Environment Variables

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

❌ DANGEROUS
const dbPassword = 'secretpass123'; // Never hardcode!
```

Performance Tips

Use Async Concurrency

Start independent operations in parallel with Promise.all().

Avoid Blocking Ops

Never use synchronous file/db operations in production.

Cache When Possible

Store computed results to avoid recalculation.

Use Streams

For large files, use streams instead of loading into memory.

🧪 Testing

Always test your code! Common testing frameworks:

- **Jest** — Full-featured testing (unit, integration)
- **Mocha** — Flexible testing framework
- **Vitest** — Fast unit test framework
- **Supertest** — HTTP assertion library

📚 Course Summary

What You Learned

- Node.js runtime and event loop
- Modules and npm package management
- Callbacks, Promises, and async/await
- File system operations
- HTTP servers and routing
- Debugging and error handling
- Best practices and security

🚀 Next Steps

Express.js

Web framework for building APIs and web apps (exp1 course).

Databases

Learn MongoDB, PostgreSQL, or other databases.

Testing

Comprehensive unit, integration, and end-to-end testing.

Deployment

Deploy to Heroku, AWS, or other cloud platforms.

💻 Coding Challenges

Challenge 1: Complete App Review

Review a complete Node.js app for best practices, security, and error handling.

Goal: Identify improvements using patterns from this course.

→ Solution

Challenge 2: Build a Mini Project

Create a complete app combining: modules, file I/O, HTTP server, error handling, logging.

Goal: Apply everything you've learned in one project.

→ Solution

Challenge 3: Production Checklist

Create a checklist of requirements for shipping a Node.js app to production.

Goal: Understand production-readiness requirements.

→ Solution

💡 Remember

Write for the reader (including future you): Code clarity matters more than cleverness.

Test everything: Bugs found in production are expensive to fix.

Monitor and log: You can't fix what you don't know about.

Keep learning: JavaScript and Node.js evolve constantly. Stay current!

📖 Learning Resources

- **Official Node.js Docs:** nodejs.org/docs
- **NPM Packages:** npmjs.com (explore before reinventing)
- **Express.js:** expressjs.com (next framework to learn)
- **Stack Overflow:** Search for answers to your questions
- **GitHub:** Study open-source Node.js projects

🎉 Congratulations!

You've Completed Node.js Fundamentals!

You now have a solid foundation in Node.js development. You understand:

- How Node.js works under the hood (event loop, async I/O)
- How to organize code with modules and packages
- How to write clean, efficient async code
- How to build HTTP servers and APIs
- How to debug and handle errors like a pro
- Best practices for production applications

Your next steps: Choose a project and build something! Learning by doing is the most effective way to deepen your skills.