Introduction to Node.js & Runtime

Generated by Pustakam Injin

AI-Powered Knowledge Engine • **gpt-oss-120b**

Pustakam Injin

AI-Powered Knowledge Creation

Document Information

Word Count	**28,859**
Chapters	**9**
Generated	**February 14, 2026**
AI Model	**Cerebras - gpt-oss-120b**

Tanmay Kalbande

Creator & Lead Architect

pustakamai.tanmaysk.in

linkedin.com/in/tanmay-kalbande

Introduction to Node.js & Runtime

Generated: 14/02/2026 **Words:** 28,859 **Provider:** Cerebras (gpt-oss-120b)

Introduction

Welcome to the World of Server-Side JavaScript

If you've spent any time building a static website, playing with HTML, CSS, or even a little bit of client-side JavaScript, you've already tasted what the web can do. What you may not have realized is that the same language that runs in the browser can also power the engines behind it-handling requests, talking to databases, authenticating users, and scaling to millions of concurrent visitors. This book is your invitation to step behind the curtain and learn **Node.js**, the JavaScript runtime that turns a simple script into a full-featured server-side application.

Whether you are a self-taught coder, a recent graduate, a hobbyist developer, or a professional looking to broaden your toolkit, this guide is written for you. No prior experience with server-side programming is required; all you need is curiosity, a willingness to type a few lines of code, and an internet connection. By the end of the journey, you'll be able to spin up a production-ready web service from scratch-using only JavaScript and a handful of powerful, open-source libraries.

Why Node.js?

The web has evolved dramatically since its inception. In the early days, servers were written in languages like Perl, PHP, or Java, each with its own learning curve and deployment quirks. Node.js arrived in 2009 with a bold promise: **"JavaScript everywhere."** By leveraging Google's V8 engine-renowned for its speed-and a non-blocking, event-driven architecture, Node made it possible to write highly concurrent servers with a language most front-end developers already know.

Here are a few reasons why learning Node.js is a smart move right now:

Benefit	What It Means for You
Unified language stack	Write both client-side and server-side code in the same language, reducing context switching and simplifying team communication.
Performance at scale	The event loop handles thousands of simultaneous connections with far less memory than traditional thread-per-request models.
Vast ecosystem	Over a million packages on npm give you instant access to tools for everything from authentication to real-time websockets.
Industry demand	Companies from startups to tech giants (Netflix, LinkedIn, Uber) rely on Node for critical services-knowing it opens doors to many careers.
Rapid prototyping	Spin up a functional API in minutes, iterate quickly, and see results instantly-perfect for MVPs and hackathons.

If any of these points sparked excitement, you're already motivated to dive deeper. This book will turn that excitement into competence.

What You'll Learn

By the time you close the final chapter, you will be able to:

1. **Explain the fundamentals of the Node.js runtime**, including the role of the event loop, the single-threaded model, and how asynchronous I/O works under the hood.
2. **Set up a clean development environment** on Windows, macOS, or Linux, and use essential tools such as `npm`, `npx`, and `nodemon`.
3. **Work with Node's core modules** (e.g., `fs`, `path`, `http`, `crypto`) and understand when to reach for them versus third-party packages.
4. **Create a basic HTTP server** from the ground up, handling requests, responses, and routing without any external frameworks.
5. **Leverage Express.js**, the de-facto web framework for Node, to build clean, modular routes, serve static assets, and manage middleware pipelines.
6. **Implement middleware for logging, parsing, error handling, and security**, learning how each piece fits into the request-response lifecycle.
7. **Integrate MongoDB using Mongoose**, defining schemas, performing CRUD operations, and handling validation and relationships.
8. **Add user authentication and authorization**, using JSON Web Tokens (JWT) and password hashing to protect your endpoints.
9. **Deploy a Node application to the cloud**, configure environment variables, set up process managers (PM2), and understand basic scaling strategies.

Each of these skills is presented in bite-size, hands-on examples that you can run, modify, and expand on your own computer. No theory without practice-every concept is reinforced with a working code snippet that you can copy-paste, run, and see the results instantly.

How the Book Is Structured

The roadmap below mirrors the natural progression of a real-world project, from

the first line of code to a live, scalable service.

1. Introduction to Node.js & Runtime

We start with a high-level overview: what Node is, why it matters, and how its event-driven architecture differs from traditional servers. You'll also get a quick tour of the V8 engine and the Node process model.

2. Setting Up the Development Environment

Step-by-step instructions for installing Node, npm, and useful utilities across all major operating systems. We'll also configure a simple project skeleton with Git version control.

3. Core Modules & the Event Loop

Dive into built-in modules, explore the event loop diagram, and write your first asynchronous code using callbacks, promises, and async/await.

4. Building a Simple HTTP Server

Skip the framework for a moment and construct a vanilla `http` server. This hands-on exercise demystifies request handling and response streaming.

5. Routing and Express.js Basics

Introduce Express, create modular routers, and learn how to separate concerns with clean folder structures.

6. Middleware, Error Handling, and Static Files

Layer your application with middleware for logging, body parsing, CORS, and static file serving. Learn to capture and respond to errors gracefully.

7. Working with Databases - MongoDB & Mongoose

Set up a local MongoDB instance (or use a cloud service), define models with Mongoose, and perform real CRUD operations.

8. Authentication & Security

Implement password hashing with bcrypt, issue JWTs, protect routes, and apply basic security headers with Helmet.

9. Deployment, Environment Variables, and Scaling

Package your app for production, deploy to a platform like Heroku or Render, manage secrets with `.env` files, and explore scaling options with PM2 and Docker.

Each chapter ends with **"Try It Yourself"** challenges-small tasks that push you to apply what you just learned, reinforcing retention and building confidence.

Setting the Right Expectations

Learning a new technology is a journey, not an instant transformation. Here's what you can realistically expect as you work through this book:

Expectation	Reality
Immediate mastery	You will gain solid foundational knowledge, but true mastery comes from building real projects beyond the examples.
All concepts covered	The book focuses on the most common patterns for web APIs. Advanced topics (e.g., GraphQL, micro-services, real-time sockets) are left for later exploration.
Zero prior backend experience needed	We start from first principles, but you should be comfortable with basic JavaScript

Expectation	Reality
	syntax and have a computer you can experiment on.
A functional app at the end	By chapter 9 you'll have a deployable, secure API that you can showcase in a portfolio or use as a foundation for larger applications.
Ongoing learning	The Node ecosystem evolves rapidly. The skills you acquire-reading documentation, debugging asynchronous code, and evaluating npm packages-will empower you to stay current.

If you ever feel stuck, remember that the Node community is famously open and supportive. The book includes links to official docs, popular tutorials, and forums where you can ask questions. Most importantly, adopt a **"learn-by-doing"** mindset: type the code, break it, fix it, and celebrate each small victory.

Your Path Forward

Think of this book as a **workshop** rather than a textbook. Every chapter is a hands-on session where you'll write, run, and test code in real time. By the time you finish, you'll have a portfolio-ready project-an API that can register users, store data in MongoDB, protect routes with JWTs, and be deployed to the cloud with a single command.

Here's a quick checklist to keep you on track:

1. **Set up your environment** before you start reading-install Node, Git, and a code editor (VS Code works great).
2. **Follow the examples verbatim** the first time; then experiment-change variable names, add extra routes, break things on purpose.
3. **Complete the "Try It Yourself" exercises** after each chapter; they are the bridge between passive reading and active mastery.

4. **Take notes** on concepts that feel new (e.g., event loop phases, middleware order). Writing them down helps solidify memory.
5. **Share your progress** on social media or a developer forum. Teaching others is the fastest way to deepen your understanding.

Remember, the goal isn't just to finish a book-it's to **gain confidence building server-side applications with JavaScript**. When you close the final page, you should feel comfortable answering questions like:

- "How does Node handle 10,000 simultaneous connections without spawning 10,000 threads?"
- "Where should I store my database credentials, and how do I access them in code?"
- "What's the simplest way to add authentication to an existing Express route?"

If you can answer those, you've succeeded.

Let's Get Started!

Take a deep breath, open your favorite editor, and fire up a terminal. In the next chapter, we'll pull back the curtain on the Node runtime itself-exploring its history, its architecture, and the philosophy that makes it a perfect fit for modern web development.

Welcome aboard this adventure into server-side JavaScript. The world of scalable, real-time web applications awaits, and you now have the map to navigate it. Let's write some code and build something amazing together.

Introduction to Node.js & Runtime

> ***"Node.js lets you use JavaScript the language of the browser to***

> *write server-side programs that run anywhere."*

In this first module we lay the foundation for everything that follows in the book. You will discover **what Node.js actually is**, why it feels familiar yet behaves very differently from the JavaScript you run in a web page, and how its core engine (Google's V8) and **event-driven, non-blocking architecture** make it uniquely suited for modern, high-throughput web services. By the end of the chapter you will be ready to write your first Node script, understand the runtime environment, and recognise the most common scenarios where server-side JavaScript shines.

Introduction

When you open a web page, the browser downloads **HTML**, **CSS**, and **JavaScript** files and executes the JavaScript inside a sandbox that has direct access to the Document Object Model (DOM). For decades that was the ***only*** environment where JavaScript lived.

Node.js, released in 2009 by Ryan Dahl, turned that assumption on its head. It packaged the V8 JavaScript engine (the same engine that powers Chrome) together with a **runtime library** that exposes operating-system resources-file system, network sockets, process management, etc.-to JavaScript code. In other words, **Node lets you write JavaScript that talks directly to the OS, just like you would with Python, Ruby, or Go**.

Why does that matter?

- **Full-stack consistency** - You can use the same language, tooling, and coding style on both the client (browser) and the server.
- **Rapid prototyping** - JavaScript's dynamic nature and massive npm ecosystem let you spin up a service in minutes.
- **Scalable I/O** - Node's event loop makes it exceptionally good at handling many

simultaneous network connections without spawning a thread per request.

In this chapter we'll unpack these ideas piece by piece, then jump straight into a hands-on example that you can run on your own machine right now.

Core Concepts

1. What is Node.js?

Aspect	Description
Runtime	A **JavaScript engine (V8)** + a **standard library** that provides APIs for file I/O, networking, child processes, etc.
Package Manager	**npm** (Node Package Manager) - the world's largest repository of reusable JavaScript modules.
Execution Model	Single-threaded **event loop** that processes callbacks and promises asynchronously.
Cross-Platform	Runs on Windows, macOS, Linux, and even on ARM devices like Raspberry Pi.
Open Source	Licensed under the MIT license; contributions from a global community.

> ***Bottom-line: Node is not a framework (like Express) nor a library; it is the environment that makes JavaScript capable of doing server-side work.***

---

2. Node.js vs. Browser JavaScript

Feature	Browser JavaScript	Node.js
Global Object	`window` (represents the browser window)	`global` (represents the Node process)
DOM Access	Full DOM API (`document`, `HTMLElement`, etc.)	No DOM - you must use libraries (e.g., `jsdom`) if needed
Modules	Historically `<script>` tags; now ES Modules (`import`) or older CommonJS (`require`)	Supports both **CommonJS** (`require`) and **ES Modules** (`import`) out of the box
File System	Not allowed (security sandbox)	Full read/write access via `fs` module
Networking	Limited to HTTP(s) requests via `fetch`, `XMLHttpRequest`, WebSockets	Direct TCP/UDP sockets, HTTP server/client, TLS, etc.
Process Control	No access to OS processes	Can spawn child processes, read environment variables, signal handling
Event Loop	Browser's event loop (similar but with extra UI tasks)	Node's own event loop (libuv) optimized for I/O
Package Management	CDN, script tags, bundlers (Webpack, Rollup)	**npm** / **yarn** - install packages from the command line

Quick Code Comparison

Browser (fetch API)

```
// In a <script> tag or a module loaded by the browser
fetch('https://api.example.com/users')
  .then(res => res.json())
  .then(users => console.log(users))
  .catch(err => console.error(err));
```

Node (http module)

```
// saved as request.js and run with `node request.js`
const https = require('https');

https.get('https://api.example.com/users', (res) => {
  let data = '';
  res.on('data', chunk => data += chunk);
  res.on('end', () => console.log(JSON.parse(data)));
}).on('error', err => console.error(err));
```

Both snippets perform an HTTP GET request, but the **Node version** works ***outside*** a browser, directly against the operating system's networking stack.

---

3. The V8 Engine - JavaScript's Speed Engine

V8 is a **high-performance JavaScript engine** written in C++. It powers Google Chrome, Chromium-based browsers, and Node.js. Its key features that matter to Node developers are:

Feature	What It Does	Why It Matters for Node
Just-In-Time (JIT) Compilation	Parses JavaScript, then compiles hot code paths to native machine code on the fly.	Your server code runs at near-native speed after a brief warm-up period.
Garbage Collection	Automatic memory management (mark-and-sweep, generational GC).	You can focus on business logic without manual memory handling.
Optimising Compiler (TurboFan)	Performs aggressive optimisations (inline caching, hidden classes).	Heavy-load APIs (e.g., JSON parsing) become extremely fast.
Ignition Interpreter	Starts execution quickly using a lightweight interpreter before TurboFan kicks in.	Fast startup times for command-line tools and micro-services.

Node's relationship to V8: Node embeds V8 as a library (`libv8`). All JavaScript you write is fed to V8 for execution, while Node's own C++ layer (via **libuv**) provides the non-blocking I/O primitives that V8 can call into as "native" functions.

> ***Takeaway: When you hear "Node is fast", it's largely because V8 does the heavy lifting of turning your JavaScript into machine code.***

---

4. Event-Driven & Non-Blocking I/O - The Heartbeat of Node

4.1 The Event Loop in a Nutshell

Node runs **one JavaScript thread**. Instead of blocking that thread for I/O (e.g., reading a file, waiting for a network response), Node registers a ***callback*** with the operating system. When the OS signals that the operation is complete, the callback is queued onto the **event loop** and executed later.

Key consequences:

- **Scalability:** One thread can handle thousands of concurrent connections because

it never waits idly.

- **Responsiveness:** CPU-bound work can still block the event loop, so you must keep callbacks short or offload heavy tasks to worker threads or child processes.
- **Predictable Concurrency Model:** No race conditions from multiple threads accessing shared memory (unless you deliberately use worker threads).

4.2 The Phases of the Event Loop

Node's event loop (implemented by **libuv**) cycles through several ***phases*** on each tick. Understanding them helps you debug performance issues.

Phase	What Happens
timers	Executes callbacks scheduled by `setTimeout()` and `setInterval()` whose delay has elapsed.
pending callbacks	Executes I/O callbacks deferred to the next loop iteration (e.g., TCP errors).
idle, prepare	Internal use; rarely visible to developers.
poll	Retrieves new I/O events; executes I/O callbacks (e.g., `fs.readFile` completion).
check	Executes callbacks scheduled by `setImmediate()`.
close callbacks	Runs `close` event callbacks (e.g., `socket.on('close', ...)`).

> ***Tip:*** *`setImmediate()` runs after I/O callbacks but before timers that have a zero-delay. This ordering is useful when you need to defer work until the current I/O cycle finishes.*

4.3 Asynchronous Patterns in Modern Node

- **Callbacks** - The original style (`fs.readFile(path, (err, data) => { ... })`).
- **Promises** - Standardised in ES2015; many Node APIs now return promises

(`fs.promises.readFile`).

- **async/await** - Syntactic sugar on top of promises; makes asynchronous code look synchronous.

```
// Using async/await with the promise-based fs API
const fs = require('fs').promises;

async function readConfig() {
  try {
    const raw = await fs.readFile('./config.json', 'utf8');
    const cfg = JSON.parse(raw);
    console.log('Config loaded:', cfg);
  } catch (e) {
    console.error('Failed to read config:', e);
  }
}
readConfig();
```

The **event loop** never "pauses" for the `await`; it simply registers the continuation (the code after `await`) as a callback to be run when the promise resolves.

---

5. Common Use-Cases for Server-Side JavaScript

Use-Case	Why Node shines	Typical Stack / Packages
Web APIs / REST services	Fast JSON handling, non-blocking I/O, abundant HTTP frameworks	Express, Fastify, Koa
Real-time applications (chat, live dashboards)	Built-in WebSocket support, low latency event loop	`ws`, Socket.io, `@socket.io/redis-adapter`
Microservices & Serverless functions	Small footprint, quick cold-start, single-file deployment	AWS Lambda (Node runtime), Azure Functions
Command-line tools & DevOps scripts	Access to OS, npm modules for everything (CLI parsing, Git, Docker)	`commander`, `yargs`, `node-fetch`

Use-Case	Why Node shines	Typical Stack / Packages
Static site generators	File system heavy, fast templating, easy integration with npm	Eleventy, Astro, Gatsby (Node-based build step)
IoT & Edge devices	Runs on low-power ARM boards, supports MQTT, BLE, GPIO via native bindings	`johnny-five`, `mqtt`, `onoff`
Data streaming & ETL pipelines	Streams API enables back-pressure handling, non-blocking transforms	`stream`, `pipeline`, `csv-parser`

> ***Bottom line:** Whenever you need high concurrency, fast JSON, or rapid prototyping, Node is a natural fit.*

Practical Application

Now that the theory is in place, let's roll up our sleeves. The following sections walk you through two hands-on exercises you can complete on any computer with Node installed (version 18+is recommended).

6. Hands-On: Your First Node Program

6.1 Install Node

1. **Download** the installer from <https://nodejs.org> (choose LTS).
2. Run the installer - it adds both `node` and `npm` to your system PATH.
3. Verify the installation:

```
$ node -v
v18.17.0
$ npm -v
9.6.7
```

6.2 Create a "Hello, World!" script

1. Open a terminal (or PowerShell / CMD).
2. Create a new folder and navigate into it:

```
mkdir node-intro && cd node-intro
```

3. Create a file called `hello.js`:

```
touch hello.js   # macOS / Linux
# or
type nul > hello.js   # Windows PowerShell
```

4. Edit `hello.js` (any editor will do) and add:

```
// hello.js - the simplest Node program
console.log('🚀 Hello, Node.js! The current time is', new Date().toLocaleTimeString());
```

5. Run the script:

```
node hello.js
```

You should see something like:

```
🚀 Hello, Node.js! The current time is 14:32:07
```

What happened?

- Node started the V8 engine, compiled `hello.js`, and executed the top-level code.
- `console.log` is a global function provided by Node (similar to the browser's `console`).

---

7. Practical Exercise: Building a Tiny HTTP Server

A **server** is the most common thing you'll write with Node. Let's create a minimal HTTP server that responds with JSON data and demonstrates the event-driven model.

7.1 Step-by-step code

Create a file named `server.js`:

```
// server.js - a tiny JSON API using only Node core modules

// 1️⃣ Import the built-in http module
const http = require('http');

// 2️⃣ Define a port (environment variable or fallback)
const PORT = process.env.PORT || 3000;

// 3️⃣ Create a request listener function
function requestHandler(req, res) {
  // Log each incoming request (demonstrates non-blocking console I/O)
  console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);

  // Simple routing - only respond to GET /api/time
  if (req.method === 'GET' && req.url === '/api/time') {
    // Set response headers for JSON
    res.setHeader('Content-Type', 'application/json');
    // Write a JSON payload
    const payload = {
      serverTime: new Date().toISOString(),
      uptime: process.uptime() // seconds the process has been alive
    };
    // End the response (non-blocking I/O under the hood)
    res.end(JSON.stringify(payload));
  } else {
    // 404 for everything else
    res.statusCode = 404;
    res.setHeader('Content-Type', 'text/plain');
    res.end('Not Found');
  }
}

// 4️⃣ Create the HTTP server instance
const server = http.createServer(requestHandler);

// 5️⃣ Start listening - this registers a non-blocking socket with libuv
server.listen(PORT, () => {
  console.log(`🚀 Server listening on http://localhost:${PORT}`);
});
```

7.2 Run the server

```
node server.js
```

You should see:

```
 Server listening on http://localhost:3000
```

Open a new terminal or your browser and request the endpoint:

```
curl http://localhost:3000/api/time
```

Result (formatted for readability):

```
{
  "serverTime": "2026-02-14T14:45:12.347Z",
  "uptime": 3.112
}
```

In the server's console you'll also see a log line for each request, proving that the **event loop** handled the incoming connection, executed `requestHandler`, and then went back to waiting for the next event.

7.3 Extending the Example

Add a **POST** endpoint that echoes back JSON data. Replace the `requestHandler` with the following (keep the rest of the file unchanged):

```
function requestHandler(req, res) {
  console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);

  // Helper to collect request body (asynchronously)
  const collectBody = () =>
    new Promise((resolve, reject) => {
      let body = '';
      req.on('data', chunk => (body += chunk));
      req.on('end', () => resolve(body));
      req.on('error', err => reject(err));
    });

  if (req.method === 'GET' && req.url === '/api/time') {
    // same as before ...
    res.setHeader('Content-Type', 'application/json');
    const payload = { serverTime: new Date().toISOString(), uptime: process.uptime() };
    res.end(JSON.stringify(payload));
  } else if (req.method === 'POST' && req.url === '/api/echo') {
    // Read the body, parse JSON, and send it back
    collectBody()
      .then(raw => {
        let data;
        try {
          data = JSON.parse(raw);
        } catch {
          res.statusCode = 400;
          return res.end('Invalid JSON');
        }
        res.setHeader('Content-Type', 'application/json');
        res.end(JSON.stringify({ received: data }));
      })
      .catch(err => {
        res.statusCode = 500;
        res.end('Server error');
        console.error('Error reading body:', err);
      });
  } else {
    res.statusCode = 404;
    res.setHeader('Content-Type', 'text/plain');
    res.end('Not Found');
```

... (continued on next page)

```
  }
}
```

Now test it:

```
curl -X POST http://localhost:3000/api/echo \
  -H "Content-Type: application/json" \
  -d '{"msg":"Hello from the client"}'
```

Expected output:

```
{
  "received": {
    "msg": "Hello from the client"
  }
}
```

What you just saw:

- The server **did not block** while waiting for the request body - it attached listeners (`'data'`, `'end'`) and let the event loop continue.
- When the body finished arriving, the promise resolved and the continuation (the `.then` block) was queued on the event loop.

---

8. Quick Debugging Tips for Beginners

Symptom	Likely Cause	How to Diagnose
Server hangs after a request	A callback performed a **blocking operation** (e.g., `fs.readFileSync`).	Look for any `*Sync` functions; replace with async equivalents.

Symptom	Likely Cause	How to Diagnose
Memory keeps growing	Unreleased references (e.g., storing each request in a global array).	Use `node --inspect` and Chrome DevTools to take heap snapshots.
Unexpected order of logs	Misunderstanding of **event loop phases** (`setTimeout` vs. `setImmediate`).	Insert timestamps and compare to the [official event loop diagram].

| **Port already in use** | Another process is bound to the same port. | Run `lsof -i :3000` (Unix) or `netstat -ano \| findstr 3000` (Windows). |

| **Callback never fires** | Error thrown **synchronously** before async registration. | Wrap code in `try/catch` or use `process.on('uncaughtException')` for debugging. |

Key Takeaways

1. **Node.js is a JavaScript runtime** built on Google's V8 engine plus a C++ library (libuv) that exposes OS capabilities to JavaScript.
2. **Browser JavaScript ≠ Node.js** - the global objects, available APIs, and module systems differ, even though the language core is the same.
3. **V8's JIT compilation** gives Node near-native speed after a brief warm-up, while its garbage collector handles memory automatically.
4. **The event loop** is the engine that makes Node non-blocking: I/O operations register callbacks, the loop processes them when ready, and the single JavaScript thread stays free to handle more work.
5. **Common server-side scenarios** where Node excels: REST/GraphQL APIs, real-time sockets, micro-services, CLI tools, static site generators, and IoT edge apps.
6. **Hands-on**: You now have a "Hello, World!" script, a tiny JSON API server, and a pattern for handling asynchronous request bodies with `async/await`.
7. **Best practices for beginners** - avoid sync APIs, keep callbacks short, use promises/async-await, and leverage the massive npm ecosystem for everything from

routing (`express`) to database drivers (`pg`, `mongoose`).

What's Next?

In **Module 2** we'll dive deeper into **npm**, **package.json**, and the **module system** that lets you organise code into reusable libraries. You'll learn how to bring third-party packages into your project, lock down versions, and write clean, maintainable Node modules.

Ready to experiment?

- Clone this repository (or create a folder) and try extending the server with a new route that reads a file from disk (`fs.promises.readFile`).
- Observe how the event loop stays responsive while the file is being read.

Congratulations - you've taken the first concrete step toward becoming a **full-stack JavaScript developer**!

Setting Up the Development Environment

> ***"A solid development environment is the launchpad for every successful Node.js project. When the tools work for you, you can focus on learning concepts instead of fighting configuration."***

In this chapter you will turn the abstract idea of "Node.js runs on a computer" into a concrete, ready-to-code workstation. We will walk through **installing Node.js**, **choosing a package manager**, **bootstrapping a project**, and **adding the tools that make development painless**-all while keeping the focus on hands-on practice. By the end you will have a working folder that you can reuse for every new backend application you build in the rest of the book.

Introduction

The first module gave you a high-level view of what Node.js is and why it matters. Now it's time to **bring that runtime onto your own machine**. The steps we'll cover are deliberately ordered to mirror the way professional teams set up fresh projects:

1. **Install Node.js** (which bundles the `node` executable and the default package manager, **npm**).
2. **Choose a package manager** (npm or Yarn) and understand the role of `package.json`.
3. **Create a new project folder** and generate a starter `package.json` with `npm init` (or `yarn init`).
4. **Add development-time helpers** - `nodemon` for automatic restarts and VS Code extensions for live linting, debugging, and IntelliSense.

Each of these steps will be demonstrated on **Windows, macOS, and Linux**, with notes on the small differences that matter. The chapter ends with a **mini-project** that you can run immediately, giving you a tangible "Hello, Node!" that you'll expand throughout the book.

> ***Why this matters - When you later learn about asynchronous I/O, streams, or server frameworks, you'll be able to test code instantly because the environment you build now will auto-restart on changes and highlight syntax errors before you even run the program.***

Core Concepts

1. Node.js, npm, and Yarn - What's the Relationship?

Concept	What it does	Typical use case
Node.js	JavaScript runtime built on	Running backend code, CLI

Concept	What it does	Typical use case
	V8. Executes `.js` files on the server.	tools, scripts.
npm	Default package manager bundled with Node. Installs packages, runs scripts, maintains `package.json`.	Most tutorials, default for new projects.
Yarn	Alternative package manager (originally by Facebook). Faster installs, deterministic lockfile (`yarn.lock`).	Teams that need strict reproducibility or prefer Yarn's workspace features.

Both npm and Yarn resolve the same **npm registry** (the public store of open-source packages). The choice is a matter of workflow preference; **the rest of the book will use npm**, but the commands for Yarn are shown side-by-side for completeness.

2. `package.json` - The Project Manifest

A `package.json` file is the **single source of truth** for a Node project. It records:

- **Metadata** - name, version, author, license.
- **Dependencies** - libraries your code needs at runtime (`express`, `dotenv`, etc.).
- **Dev-dependencies** - tools used only during development (`nodemon`, `eslint`).
- **Scripts** - shortcut commands (`npm start`, `npm test`) that can invoke any shell command.
- **Engines** - optional constraints on the Node version (`"node": ">=14"`).

When you run `npm install <pkg> --save`, npm automatically writes the dependency into `package.json`. Conversely, `npm install` (no arguments) reads the manifest and installs every listed package into `node_modules`.

3. Version Managers - Keeping Node Versions Tidy

If you ever need to jump between projects that require different Node versions (e.g., one uses Node 12, another Node 18), a **version manager** saves you from

reinstalling the entire runtime each time.

Manager	OS Support	Quick Install Command
nvm-windows	Windows	Download installer from <https://github.com/coreybutler/nvm-windows/releases>

| **nvm** (Node Version Manager) | macOS, Linux | `curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.5/install.sh \| bash` | | **fnm** (Fast Node Manager) | macOS, Linux, Windows (via WSL) | `brew install fnm` (macOS) or `curl -fsSL https://fnm.vercel.app/install \| bash` |

You **don't have to use a version manager** for this book, but the short sections below will show you how to install one, because many learners eventually need it.

4. `nodemon` - Auto-Restarting the Server

When you edit a server file, you normally have to stop the running process (`Ctrl+C`) and start it again (`node index.js`). `nodemon` watches the filesystem and automatically restarts the process whenever a file changes. This reduces friction dramatically, especially when learning asynchronous concepts where you want rapid feedback.

Installed as a dev-dependency:

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

Typical script entry:

```
"scripts": {
  "dev": "nodemon index.js"
}
```

Running `npm run dev` now starts your server and watches for changes.

5. Visual Studio Code (VS Code)- The IDE of Choice

VS Codeis free, cross-platform, and has a **rich ecosystem of extensions** that make Node development smoother:

Extension	Purpose	Install Command (inside VS Code)
ESLint	Real-time linting for JavaScript/Node	`ext install dbaeumer.vscode-eslint`
Prettier - Code formatter	Automatic code formatting on save	`ext install esbenp.prettier-vscode`
Node.js Extension Pack	Bundles debugger, npm scripts view, snippets	`ext install waderyan.nodejs-extension-pack`
Debugger for Chrome (or built-in Node debugger)	Set breakpoints, step through async code	Built-in, enable "Node" debug configuration
GitLens	Visualize Git history, blame, and diff	`ext install eamodio.gitlens`

These extensions will be briefly configured in the **Practical Application** section.

Practical Application

Below is a **step-by-step, cross-platform tutorial**. Follow it on your own machine; the commands are copy-and-paste ready. If you hit an error, the "Troubleshooting" boxes after each major step provide quick fixes.

1. Install Node.js (with npm)

1.1 Windows

1. Visit the official download page: <https://nodejs.org/en/download/>.
2. Download the **Windows Installer (MSI)** for the **LTS** version (currently 20.x).

3. Run the installer:

- Accept the license.
- Choose the default install location (e.g., `C:\Program Files\nodejs`).
- **Important:** Tick **"Add to PATH"** - this makes `node` and `npm` available from any command prompt.
- Keep the **npm package manager** option selected.

4. Finish the wizard and open a **new** Command Prompt (or PowerShell).

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

Both commands should print version numbers (e.g., `v20.12.0` and `9.8.1`). If you get "command not found", close and reopen the terminal.

> ***Troubleshooting (Windows) - If PATH wasn't added, you can manually edit System Properties → Environment Variables → Path and add `C:\Program Files\nodejs\`.***

1.2 macOS

Two common ways: **Homebrew** (recommended) or the **macOS Installer**.

Homebrew method (if you already have Homebrew):

```
brew update
brew install node   # Installs latest LTS + npm
```

Installer method:

1. Go to <https://nodejs.org/en/download/> and click ***macOS Installer (.pkg)*** for LTS.
2. Run the `.pkg` file and follow the prompts.
3. After installation, open **Terminal** and verify:

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

> *Troubleshooting (macOS) - If `node` is not found, you may need to restart the terminal or run `export PATH="/usr/local/bin:$PATH"` (for Intel) or `export PATH="/opt/homebrew/bin:$PATH"` (for Apple Silicon) in `~/.zshrc`.*

1.3 Linux

Most distributions ship a fairly recent Node version in their repositories, but it may lag behind the official LTS. Use the **NodeSource** binary distribution for the latest LTS.

```
# Ubuntu/Debian
curl -fsSL https://deb.nodesource.com/setup_lts.x | sudo -E bash -
sudo apt-get install -y nodejs   # Installs node and npm

# Verify
node -v
npm -v
```

For **Fedora**:

```
curl -fsSL https://rpm.nodesource.com/setup_lts.x | sudo bash -
sudo dnf install -y nodejs
node -v && npm -v
```

> *Troubleshooting (Linux) - If you see "npm: command not found", install the `npm` package separately (`sudo apt-get install npm`). On Alpine Linux, use `apk add nodejs npm`.*

2. (Optional) Install a Node Version Manager

If you anticipate needing multiple Node versions, install **nvm** (macOS/Linux) or **nvm-windows**.

macOS/Linux - nvm:

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.5/install.sh | bash
# Activate in current shell
export NVM_DIR="$([ -z "${XDG_CONFIG_HOME-}" ] && printf %s "${HOME}/.nvm" || printf %s
"${XDG_CONFIG_HOME}/nvm")"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh"
# Install latest LTS
nvm install --lts
# Use it
nvm use --lts
node -v   # Should show the LTS version you just installed
```

Windows - nvm-windows:

1. Download the latest installer from <https://github.com/coreybutler/nvm-windows/releases>.
2. Run the installer (default options are fine).
3. Open a **new** PowerShell window:

```
nvm install lts
nvm use lts
node -v
```

Now you can switch versions with `nvm use <version>`.

3. Choose a Package Manager (npm vs Yarn)

For the remainder of this book we will **use npm**, but if you prefer Yarn, replace the commands as shown.

Install Yarn (optional) - Yarn can be installed globally via npm:

```
npm install -g yarn
yarn -v   # Should print a version like 1.22.x or 3.x
```

> ***Note - Yarn 2+ (aka "Berry") changes the default folder layout and requires a `yarnrc.yml`. Stick with Yarn 1 (`npm i -g yarn`) if you want the classic experience.***

4. Initialise a New Node Project

Create a fresh folder for the demo project:

```
mkdir node-hello-world
cd node-hello-world
```

Now run the interactive initializer:

```
npm init
```

You'll be prompted:

```
package name: (node-hello-world)
version: (1.0.0)
description: A minimal Node server for Module 2 exercises
entry point: (index.js)
test command:
git repository:
keywords: node, hello, tutorial
author: Your Name <you@example.com>
license: (ISC)
```

Press **Enter** to accept defaults, or type your own values. At the end a `package.json` file appears:

```
{
  "name": "node-hello-world",
  "version": "1.0.0",
  "description": "A minimal Node server for Module 2 exercises",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "node",
    "hello",
    "tutorial"
  ],
  "author": "Your Name <you@example.com>",
  "license": "ISC"
}
```

> *Tip - You can skip the interactive prompts with `npm init -y`, which creates a `package.json` with default values instantly.*

5. Add a Simple Server File

Create `index.js` in the project root:

```
touch index.js   # macOS/Linux
# or
type nul > index.js   # Windows PowerShell
```

Open it in VS Code (`code .` from the terminal opens the folder). Paste the following code:

```
// index.js
// A tiny HTTP server that responds with "Hello, Node!"
// Demonstrates that Node can listen on a port and send a response.

const http = require('http');

const PORT = process.env.PORT || 3000; // Use an env var if provided

const server = http.createServer((req, res) => {
  // Log each request to the console (helps while learning)
  console.log(`${req.method} ${req.url}`);

  // Set a simple JSON response header
  res.setHeader('Content-Type', 'application/json');

  // Send a JSON payload
  const payload = { message: 'Hello, Node! ' };
  res.end(JSON.stringify(payload));
});

server.listen(PORT, () => {
  console.log(` Server listening on http://localhost:${PORT}`);
});
```

> ***Explanation -***
>
> **** `http.createServer` creates a low-level server (no Express yet).***
>
> **** `process.env.PORT` lets you override the port with an environment variable-useful for cloud platforms.***
>
> **** The server logs each request, which is a great way to see the asynchronous event loop in action.***

6. Run the Server Manually

```
node index.js
```

You should see:

```
 Server listening on http://localhost:3000
```

Open a browser (or use `curl`) to test:

```
curl http://localhost:3000
```

Result:

```
{"message":"Hello, Node! "}
```

You have a **working backend**!

7. Install and Configure `nodemon`

Stop the server (`Ctrl+C`). Now add `nodemon` as a development dependency:

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

Add a **dev script** to `package.json`:

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

Run the new script:

```
npm run dev
```

You'll see something similar to:

```
[nodemon] 2.0.22
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
 Server listening on http://localhost:3000
```

Now **edit `index.js`** (e.g., change the message to "Hello, Node.js + nodemon!"). Save the file; `nodemon` automatically restarts the server and prints:

```
[nodemon] restarting due to changes...
 Server listening on http://localhost:3000
```

Refresh the browser - the new message appears instantly. No manual restarts needed.

8. Set Up VS Codefor a Smooth Workflow

1. **Open the project folder** in VS Code(`code .`).
2. **Install recommended extensions** (the IDE will prompt you when it detects a `package.json`). Click **Install** for:

- ESLint
- Prettier - Code formatter
- Node.js Extension Pack

If you don't see the prompt, open the Extensions view (`Ctrl+Shift+X`) and search for the names above.

3. **Configure ESLint & Prettier** - Create a `.eslintrc.json` file:

```
{
  "env": {
    "node": true,
    "es2022": true
  },
  "extends": "eslint:recommended",
  "parserOptions": {
    "ecmaVersion": 2022
  },
  "rules": {
    "no-console": "off",
    "semi": ["error", "always"]
  }
}
```

And a `.prettierrc`:

```
{
  "singleQuote": true,
  "trailingComma": "es5",
  "printWidth": 80
}
```

4. **Enable format on save** - Open Settings (`Ctrl+,`), search for ***format on save***, and check the box. Ensure the default formatter is set to **Prettier - Code formatter**.

5. **Create a launch configuration** for debugging:

Open the Run & Debug view (`Ctrl+Shift+D`) → click "create a launch.json file" → select "Node.js".

Replace the generated config with:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Launch index.js",
      "program": "${workspaceFolder}/index.js",
      "restart": true,
      "console": "integratedTerminal"
    }
  ]
}
```

Now you can set breakpoints in `index.js`, press **F5**, and step through the code-perfect for visualising the event loop later.

9. Exercise: Extend the Server with a Simple API

Goal: Add a second route (`/time`) that returns the current server time in ISO format.

Steps:

1. Open `index.js`.
2. Inside the request handler, add a conditional branch:

```
const url = new URL(req.url, `http://${req.headers.host}`);
if (url.pathname === '/time') {
  const now = new Date().toISOString();
  res.end(JSON.stringify({ time: now }));
  return;
}
// existing hello route falls through
```

3. Save the file. `nodemon` restarts automatically.
4. Test in terminal:

```
curl http://localhost:3000/time
```

Expected output:

```
{"time":"2026-02-14T12:34:56.789Z"}
```

5. **Optional**: Open the Debugger (F5), place a breakpoint on the line `const now = new Date().toISOString();`, hit `/time` in the browser, and watch the variables.

Reflection - You just added a second endpoint without touching any external libraries. The pattern of inspecting `req.url` and returning JSON is the foundation upon which **Express**, **Koa**, or **Fastify** build their routing abstractions (which we'll explore in later modules).

10. Version-Control Quick-Start (Git)

Even for a simple tutorial, committing early builds good habits.

```
git init
git add .
git commit -m "Initial commit - hello server with nodemon & VS Code config"
```

You now have a local repository. If you have a GitHub account, create a remote repo and push:

```
git remote add origin https://github.com/yourusername/node-hello-world.git
git branch -M main
git push -u origin main
```

The next modules will ask you to **branch** for new features, so this repository will serve as the baseline.

Troubleshooting Box

Symptom	Likely Cause	Fix
`node` command not found after install	PATH not updated	Restart terminal or manually add Node's install directory to PATH (see OS-specific notes).
`npm install` hangs or fails with **E401**	Registry authentication issue (rare on fresh install)	Run `npm config delete registry` then retry.
`nodemon` does not restart on file changes	Glob pattern mismatch (e.g., you renamed the file)	Ensure the script points to the correct entry file (`nodemon index.js`).
VS Code shows "ESLint: No configuration found"	Missing `.eslintrc.*` file	Create one (see step 8) or run `npm init @eslint/config`.
Debugger stops at "Cannot find module 'node:fs'"	Node version too old for built-in `node:` imports (introduced in v14)	Upgrade Node via nvm (`nvm install --lts`).

Key Takeaways

- **Node.js + npm** are installed together; they give you the runtime and a package manager out of the box.
- **`package.json`** is the heartbeat of any Node project-keep it under version control.
- **Version managers** (nvm, nvm-windows) let you switch Node versions without reinstalling.
- **`nodemon`** eliminates manual restarts, dramatically speeding up the edit-run cycle.
- **VS Code extensions** (ESLint, Prettier, Node Debugger) provide instant feedback, consistent formatting, and powerful debugging-all essential for mastering asynchronous patterns later.
- The **hands-on mini-project** (`index.js` + a `/time` endpoint) demonstrates a functional server, environment variables, and a basic routing technique that will evolve into full-featured frameworks.

With this environment ready, you are now equipped to **focus on learning Node's core concepts**-event loop, callbacks, promises, and streams-without being held back by tooling friction. In the next module we'll dive into **asynchronous I/O**, using the server you just built as a playground. Happy coding!

Core Modules & the Event Loop

> ***"Understanding the built-in modules and the invisible engine that drives them - the event loop - is the turning point where you move from writing tiny scripts to building real-world, high-performance server-side applications."***

In this third module we dive deep into the heart of Node.js: the **core (built-in) modules** that give you file-system access, path manipulation, HTTP servers, and an event-driven architecture - and the **event loop**, the runtime mechanism that makes all of this asynchronous magic possible. By the end you will be comfortable:

- Importing and using `fs`, `path`, `http`, and `events`.
- Describing each phase of the event loop and how Node decides what to run next.
- Writing callbacks, promises, and `async/await` code that interact with the core modules.

Everything builds on what you learned in Modules 1 & 2 (what Node is and how to set up your environment). Grab a terminal, open a fresh project folder, and let's get our hands dirty.

Introduction

When you write JavaScript in the browser you are limited to the DOM, the `fetch` API, and a handful of browser-provided utilities. Node.js, on the other hand, ships with **over 30 core modules** written in C/C++ and exposed to JavaScript. They

give you direct access to the operating system - reading files, spawning processes, handling network sockets - without ever leaving the comfort of JavaScript.

But why do these modules ***feel*** asynchronous? Why can you start reading a file, continue executing other code, and later receive the file's contents? The answer lies in the **event loop**, a single-threaded orchestrator that schedules callbacks, promises, and I/O events.

In this chapter we will:

1. **Explore four core modules** that you will use in almost every backend project:
- `fs` - file system operations.
- `path` - cross-platform path utilities.
- `http` - creating web servers and clients.
- `events` - the EventEmitter class, the basis of many Node APIs.

2. **Dissect the event loop** - its phases, its queues, and how it interacts with the operating system's kernel.

3. **Apply the concepts** by building a tiny static-file web server that demonstrates callbacks, promises, and `async/await` side-by-side.

4. **Reflect on key takeaways** that you can immediately use in your own projects.

Core Concepts

1. The Four Core Modules You'll Use Everyday

Module	Primary Purpose	Typical Use-Cases	Synchronous vs Asynchronous APIs
`fs`	Interact with the file system (read/ write files,	Reading configuration files, serving static assets, logging, file	Both. Every operation has a sync version (`fs.readFileSync`) **and**

Module	Primary Purpose	Typical Use-Cases	Synchronous vs Asynchronous APIs
	directories, stats).	uploads.	an async version (`fs.readFile`).
`path`	Manipulate file-system paths in a platform-agnostic way.	Building file URLs, normalizing user input, joining directory names.	**Only** synchronous (pure string manipulation).
`http`	Create HTTP servers and make HTTP client requests.	REST APIs, static file servers, proxy servers, consuming external APIs.	Server callbacks are synchronous; client request methods (`http.get`, `http.request`) are asynchronous.
`events`	Event-emitter pattern - objects that emit named events and listeners that react.	Custom APIs, stream handling, process events (`process.on('exit')`).	Always asynchronous when you emit an event via `emit`.

> ***Pro tip: If a module offers both sync and async methods, always prefer the async version in production code. The sync version blocks the event loop, preventing other work from happening and hurting scalability.***

Below we explore each module in more depth, with code snippets that you can copy into `example.js` and run with `node example.js`.

1.1 The `fs` Module - Working with Files

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

// ---------- Asynchronous read (callback) ----------
fs.readFile(path.join(__dirname, 'data.txt'), 'utf8', (err, data) => {
  if (err) return console.error('❌ read error:', err);
  console.log('✅ Async read (callback):', data);
});

// ---------- Asynchronous read (Promise) ----------
const { promises: fsp } = fs; // shortcut to fs.promises

fsp.readFile(path.join(__dirname, 'data.txt'), 'utf8')
  .then(data => console.log('✅ Async read (Promise):', data))
  .catch(err => console.error('❌ promise error:', err));

// ---------- Synchronous read ----------
try {
  const data = fs.readFileSync(path.join(__dirname, 'data.txt'), 'utf8');
  console.log('✅ Sync read:', data);
} catch (err) {
  console.error('❌ sync error:', err);
}
```

What you see:

- `fs.readFile` uses a **callback** - the classic Node style.
- `fs.promises.readFile` returns a **Promise**, enabling `then/catch` or `await`.
- `fs.readFileSync` blocks the event loop until the file is fully read.

> ***Exercise 1: Create a file called `data.txt` with any text. Run the script and observe the order of the three logs. Which one appears first? Why?***

1.2 The `path` Module - Platform-Neutral Path Handling

```
// path-demo.js
const path = require('path');

// Join parts of a path - works on Windows, macOS, Linux
const fullPath = path.join(__dirname, 'public', 'images', 'logo.png');
console.log('📁 Joined path:', fullPath);

// Resolve a path to an absolute path
const absolute = path.resolve('src', '..', 'config', 'settings.json');
console.log('📍 Resolved absolute path:', absolute);

// Extract parts
console.log('📂 dirname:', path.dirname(fullPath));
console.log('📄 basename:', path.basename(fullPath));
console.log('🔖 extname:', path.extname(fullPath));
```

`path` does **not** touch the file system; it merely manipulates strings. Therefore it is safe to use in any part of your code, even inside performance-critical loops.

> ***Exercise 2: Run the script on your machine (Windows vs macOS vs Linux) and note how the separator (`\` vs `/`) changes automatically.***

1.3 The `http` Module - Building a Minimal Web Server

```
// http-demo.js
const http = require('http');
const fs = require('fs');
const path = require('path');

const PORT = 3000;

const server = http.createServer((req, res) => {
  // Simple routing
  if (req.url === '/' || req.url === '/index.html') {
    const indexPath = path.join(__dirname, 'public', 'index.html');
    // Async file read with stream (best for large files)
    const readStream = fs.createReadStream(indexPath);
    res.writeHead(200, { 'Content-Type': 'text/html' });
    readStream.pipe(res);
  } else {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('❌ Not Found');
  }
});

server.listen(PORT, () => {
  console.log(`🚀 Server listening on http://localhost:${PORT}`);
});
```

Key points:

- `http.createServer` receives a **request listener** (`(req, res) => {}`) that is invoked ***every time*** a new HTTP request arrives.
- Inside the listener we **asynchronously** stream a file to the response using `fs.createReadStream`. Streaming avoids loading the whole file into memory.
- The server runs **non-blocking** - while it serves a large file, it can still accept new connections.

> ***Exercise 3: Create a `public/index.html` file with any HTML content, start the server with `node http-demo.js`, and visit `http://localhost:3000`. Change the file while the server is running and refresh - notice the changes appear instantly.***

1.4 The `events` Module - EventEmitter Basics

```
// events-demo.js
const { EventEmitter } = require('events');

class Clock extends EventEmitter {
  start(intervalMs = 1000) {
    this.timer = setInterval(() => {
      this.emit('tick', new Date());
    }, intervalMs);
  }
  stop() {
    clearInterval(this.timer);
    this.emit('stopped');
  }
}

const myClock = new Clock();

myClock.on('tick', (now) => {
  console.log('⏰ Tick at', now.toLocaleTimeString());
});

myClock.once('stopped', () => {
  console.log('🛑 Clock stopped');
});

myClock.start(2000); // tick every 2 seconds

// Stop after 7 seconds
setTimeout(() => myClock.stop(), 7000);
```

- `EventEmitter` is the **foundation** of many Node APIs (`fs` streams, `http` server, `process`, etc.).
- Listeners can be added with `on` (multiple times) or `once` (auto-removed after first call).
- Emitting an event (`emit`) **queues** the listener callbacks to be run on the next tick of the event loop (more on that later).

> ***Exercise 4: Modify the `Clock` class to emit a `'minute'` event every time the seconds are `00`. Use `on('minute', ...)` to log a special message.***

2. The Event Loop - Node's Single-Threaded Conductor

2.1 Why a "Loop"?

JavaScript in the browser is famously **single- threaded** . Node follows the same model: **one JavaScript thread** executes your code, while the heavy lifting (disk I/O, network sockets, cryptography) is delegated to the **libuv thread pool** and the operating system's kernel. The event loop continuously checks for **ready callbacks** and executes them, one after another, until there's nothing left to do.

Think of the event loop as a **busy chef**:

- The chef (JavaScript thread) prepares dishes (executes callbacks).
- The dishwasher (libuv thread pool) handles washing pots (disk/network I/O).
- When a pot is clean, the dishwasher notifies the chef, who then picks up the next order.

If the chef decides to **stand still and read a cookbook** (i.e., run a blocking, synchronous operation), no other orders can be processed - the restaurant stalls.

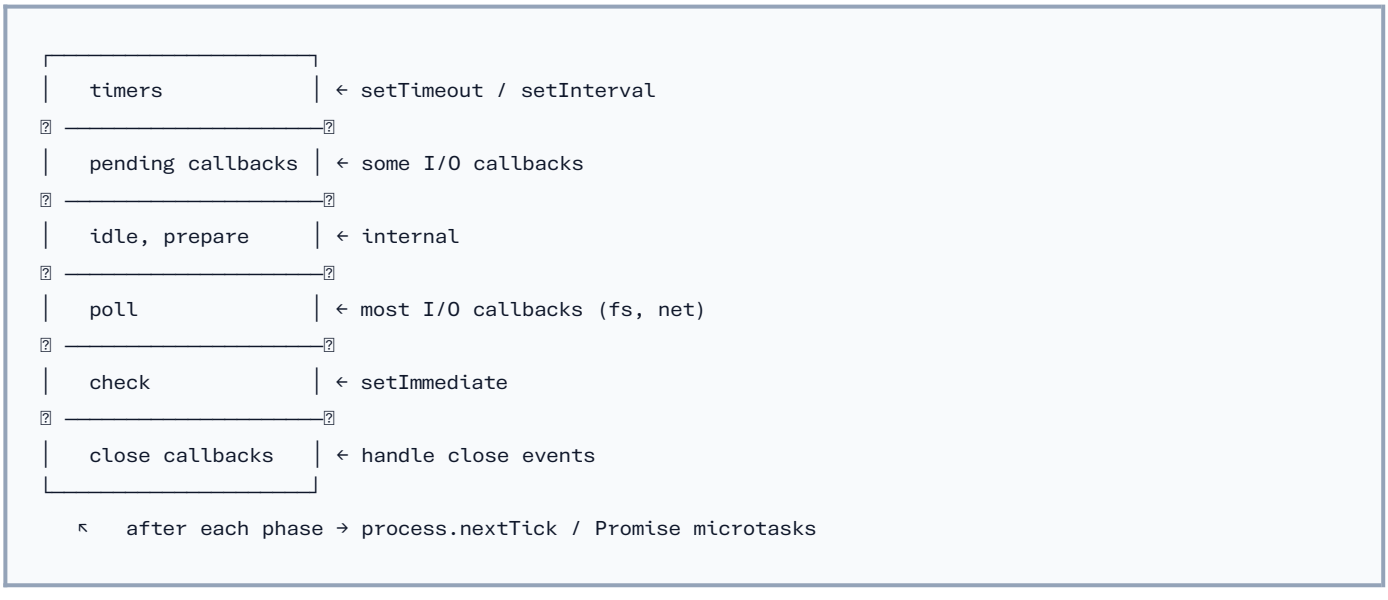
2.2 Phases of the Event Loop (Node v20+)

Node's event loop is defined by **libuv** and consists of several distinct phases. Each phase has its own queue of callbacks. The loop proceeds in order, processing ***all*** callbacks in a phase before moving to the next. Below is a simplified view that is sufficient for beginners:

Phase	What it handles	Typical callbacks
timers	Executes callbacks scheduled by `setTimeout` and `setInterval` whose timeout has elapsed.	`setTimeout(() => {}, 0)`, `setInterval`
pending callbacks	Executes I/O callbacks that were deferred to the next loop iteration.	TCP errors, some `fs` callbacks

Phase	What it handles	Typical callbacks
idle, prepare	Internal use (rarely exposed).	-
poll	Retrieves new I/O events; executes I/O callbacks (e.g., `fs.readFile` completion). If the poll queue is empty, the loop will **wait** for callbacks (blocking) or move on if there's a timer scheduled.	`fs.readFile` callback, `http` request callbacks
check	Executes callbacks scheduled by `setImmediate`.	`setImmediate(() => {})`
close callbacks	Executes callbacks for closed handles (e.g., `socket.on('close')`).	`socket.on('close')`
process.nextTick queue (microtasks)	Not a separate phase, but **runs after each phase** before the event loop proceeds to the next phase. Handles `process.nextTick` and **Promise microtasks** (`.then`, `await`).	`process.nextTick()`, `Promise.resolve().then()`

Visual representation



Key takeaways from the table

- **`setTimeout` vs `setImmediate`** - Both schedule callbacks for "later", but they run in different phases. `setImmediate` always runs **after** I/O callbacks (poll phase) and **before** timers of the next loop tick.
- **`process.nextTick` and Promises** - These are **microtasks** that run **immediately after the current operation**, before the event loop moves on. They can starve the loop if misused (e.g., an infinite `nextTick` loop).
- **I/O callbacks** - When you call `fs.readFile`, Node hands the request to libuv's thread pool. Once the file is read, libuv pushes the callback onto the **poll** queue. The next time the loop reaches the poll phase, your callback will fire.

2.3 Asynchronous I/O - How the Kernel Helps

When you call an asynchronous method (e.g., `fs.readFile`), Node performs these steps:

1. **JavaScript thread** (event loop) calls the native C++ binding.
2. The binding **queues** the operation to libuv's **thread pool** (default size = 4, configurable via `UV_THREADPOOL_SIZE`).
3. A worker thread **executes the blocking system call** (`read`, `open`, etc.).
4. Once the OS finishes the operation, the worker thread **places the callback** onto the appropriate libuv queue (`poll`).
5. The **event loop** reaches the poll phase, sees the callback, and runs it on the JavaScript thread.

Because the heavy work happens on a separate thread, the main JavaScript thread never blocks, and the server can continue handling other connections.

> ***Memory tip: If you have many simultaneous file reads, you may need to increase the thread-pool size (`UV_THREADPOOL_SIZE=8 node app.js`) or use streaming (which reuses a single thread per stream).***

2.4 Microtasks vs Macrotasks - Why `await` Feels "Instant"

When you write:

```
async function demo() {
  console.log('A');
  await Promise.resolve();
  console.log('B');
}
demo();
console.log('C');
```

The output is `A C B`. Why?

- `await` pauses the async function and **creates a Promise microtask**.
- The microtask is queued **immediately after the current stack** (i.e., after `demo()` returns).
- The event loop finishes the current **macrotask** (the top-level script), then processes all microtasks before moving to the next phase.

Thus `C` prints before `B`, even though the awaited promise resolved instantly.

---

3. Practical Application - Build a Tiny Static-File Server

Now that we know the core modules and the event loop, let's combine them into a **real-world example**: a static-file web server that serves any file from a `public/` directory. We will implement three variants of the request handler to illustrate:

1. **Callback-based** (`fs.readFile` + callbacks).
2. **Promise-based** (`fs.promises.readFile`).
3. **`async/await` with streaming** (`fs.createReadStream`).

3.1 Project Setup

```
mkdir static-server
cd static-server
npm init -y          # creates a minimal package.json
mkdir public
echo "<h1>Hello from Node!</h1>" > public/index.html
```

Create a file `server.js` and copy the code sections below. Run the server with:

```
node server.js
```

Open `http://localhost:4000` in a browser. Change the contents of `public/index.html` and refresh - the server will reflect the changes instantly.

3.2 Variant 1- Callbacks (Classic Node)

```
// server.js - Variant 1: Callbacks
const http = require('http');
const fs = require('fs');
const path = require('path');

const PORT = 4000;

function serveFileCallback(filePath, res) {
  // 1️⃣ Resolve the absolute path
  const absPath = path.resolve(filePath);

  // 2️⃣ Read file asynchronously (callback style)
  fs.readFile(absPath, (err, data) => {
    if (err) {
      // 3️⃣ If file not found, send 404
      if (err.code === 'ENOENT') {
        res.writeHead(404, { 'Content-Type': 'text/plain' });
        return res.end('❌ Not found');
      }
      // 4️⃣ For any other error, send 500
      res.writeHead(500, { 'Content-Type': 'text/plain' });
      return res.end('⚠️ Server error');
    }

    // 5️⃣ Determine MIME type (very naive)
    const ext = path.extname(absPath);
    const mime = ext === '.html' ? 'text/html' : 'application/octet-stream';

    // 6️⃣ Send successful response
    res.writeHead(200, { 'Content-Type': mime });
    res.end(data);
  });
}

const server = http.createServer((req, res) => {
  // Normalise URL (remove query string)
  const urlPath = req.url.split('?')[0];
  const safePath = urlPath === '/' ? '/index.html' : urlPath;

  // Prevent directory traversal attacks
```

... (continued on next page)

```
  const normalized = path.normalize(safePath).replace(/^(\.\.[\/\\])+/, '');
  const filePath = path.join(__dirname, 'public', normalized);

  serveFileCallback(filePath, res);
});

server.listen(PORT, () => {
  console.log(`🚀 Callback server listening on http://localhost:${PORT}`);
});
```

What you learn

- **Callback nesting** - each asynchronous step receives a callback.
- **Error handling** - we check `err.code` for `ENOENT` (file not found).
- **MIME detection** - a naïve switch; in real apps you'd use the `mime` package.

> ***Exercise 5: Add support for `.css` and `.js` files by extending the MIME detection logic.***

3.3 Variant 2- Promises (`fs.promises`)

```
// server.js - Variant 2: Promises
const http = require('http');
const { promises: fsp } = require('fs');
const path = require('path');

const PORT = 4000;

async function serveFilePromise(filePath, res) {
  try {
    const data = await fsp.readFile(filePath);
    const ext = path.extname(filePath);
    const mime = ext === '.html' ? 'text/html' : 'application/octet-stream';
    res.writeHead(200, { 'Content-Type': mime });
    res.end(data);
  } catch (err) {
    if (err.code === 'ENOENT') {
      res.writeHead(404, { 'Content-Type': 'text/plain' });
      return res.end('❌ Not found');
    }
    res.writeHead(500, { 'Content-Type': 'text/plain' });
    res.end('❌ Server error');
  }
}

const server = http.createServer((req, res) => {
  const urlPath = req.url.split('?')[0];
  const safePath = urlPath === '/' ? '/index.html' : urlPath;
  const normalized = path.normalize(safePath).replace(/^(\.\.[\/\\])+/, '');
  const filePath = path.join(__dirname, 'public', normalized);
  serveFilePromise(filePath, res);
});

server.listen(PORT, () => {
  console.log(`🚀 Promise server listening on http://localhost:${PORT}`);
});
```

Why this version is nicer

- **Linear flow** - `await` makes the code read top-to-bottom, eliminating callback "pyramid of doom".
- **Automatic microtask handling** - the `await` creates a Promise microtask that runs after the current tick, preserving the event-loop order.

> ***Exercise 6: Refactor the MIME detection into a small helper function `getMime(ext)` and reuse it.***

3.4 Variant 3- `async/await` with Streams (Best for Large Files)

```
// server.js - Variant 3: Streaming
const http = require('http');
const fs = require('fs');
const path = require('path');

const PORT = 4000;

function getMime(ext) {
  const map = {
    '.html': 'text/html',
    '.css':  'text/css',
    '.js':   'application/javascript',
    '.json': 'application/json',
    '.png':  'image/png',
    '.jpg':  'image/jpeg',
    '.svg':  'image/svg+xml',
    '.txt':  'text/plain',
  };
  return map[ext] || 'application/octet-stream';
}

async function serveFileStream(filePath, res) {
  // 1 Check existence + get stats (size, type) - use promises for consistency
  try {
    const stats = await fs.promises.stat(filePath);
    if (stats.isDirectory()) {
      // If directory, try to serve index.html inside it
      return serveFileStream(path.join(filePath, 'index.html'), res);
    }

    const ext = path.extname(filePath);
    const mime = getMime(ext);
    res.writeHead(200, {
      'Content-Type': mime,
      'Content-Length': stats.size, // helps browsers show progress
    });

    // 2 Create a read stream and pipe directly to response
    const readStream = fs.createReadStream(filePath);
    readStream.pipe(res);
```

... (continued on next page)

```
    // 3⃣ Error handling on the stream (e.g., permission denied)
    readStream.on('error', (err) => {
      console.error('⃣ Stream error:', err);
      if (!res.headersSent) {
        res.writeHead(500, { 'Content-Type': 'text/plain' });
        res.end('⃣ Stream error');
      } else {
        // If headers already sent, we can't change status - just destroy the connection
        res.destroy();
      }
    });
  } catch (err) {
    // File not found or other FS error
    if (err.code === 'ENOENT') {
      res.writeHead(404, { 'Content-Type': 'text/plain' });
      return res.end('⃣ Not found');
    }
    res.writeHead(500, { 'Content-Type': 'text/plain' });
    res.end('⃣ Server error');
  }
}

const server = http.createServer((req, res) => {
  const urlPath = decodeURIComponent(req.url.split('?')[0]);
  const safePath = urlPath === '/' ? '/index.html' : urlPath;
  const normalized = path.normalize(safePath).replace(/^(\.\.[\/\\])+/, '');
  const filePath = path.join(__dirname, 'public', normalized);
  serveFileStream(filePath, res);
});

server.listen(PORT, () => {
  console.log(`⃣ Streaming server listening on http://localhost:${PORT}`);
});
```

Advantages of streaming

- **Constant memory footprint** - only a small chunk of the file lives in memory at any time.
- **Back-pressure** - the underlying TCP socket tells the stream when to pause, preventing overwhelming the client.
- **Better for large assets** - images, videos, zip files, etc.

> ***Exercise 7: Add support for range requests (`Accept-Ranges` header) so browsers can resume partially downloaded files. (Hint: inspect the `Range` header, use `fs.createReadStream(start, end)`.)***

3.5 Observing the Event Loop in Action

Node ships with a diagnostic tool called **`node --inspect`** and a built-in **`performance`** API. Let's instrument our streaming server to log when each phase runs.

```
// add at the top of server.js
const { performance, PerformanceObserver } = require('perf_hooks');

// Observe async hooks (microtasks)
const obs = new PerformanceObserver((list) => {
  const entries = list.getEntries();
  entries.forEach((e) => console.log(`⏱ ${e.name}: ${e.duration.toFixed(2)}ms`));
});
obs.observe({ entryTypes: ['measure'] });

// Helper to mark phases
function mark(name) {
  performance.mark(`${name}-start`);
}
function measure(name) {
  performance.measure(name, `${name}-start`);
}

// Example usage inside request handler
http.createServer((req, res) => {
  mark('request-start');

  // ... normal handling ...

  res.on('finish', () => {
    measure('request-start');
    console.log('⏱ Response finished for', req.url);
  });
});
```

When you make a request, you'll see timing logs that correspond to **macro-tasks**

(the request handler) and **micro-tasks** (the Promise resolution inside `await`). Running `node --trace-event-loop` (Node v20+) prints an even more granular view, showing each phase entry and exit.

> ***Experiment: Run the server with `node --trace-event-loop server.js` and scroll through the output. Identify where the poll phase occurs when the file stream emits data.***

3.6 Bonus: Using `EventEmitter` to Decouple Logic

Let's refactor the streaming version so the request handler **emits** events (`'fileServed'`, `'error'`) instead of directly writing to the response. This demonstrates how many core modules already inherit from `EventEmitter`.

```
// server-emitter.js
const http = require('http');
const fs = require('fs');
const path = require('path');
const { EventEmitter } = require('events');

class FileServer extends EventEmitter {
  constructor(root) {
    super();
    this.root = root;
  }

  async serve(filePath, res) {
    try {
      const stats = await fs.promises.stat(filePath);
      if (stats.isDirectory()) {
        return this.serve(path.join(filePath, 'index.html'), res);
      }

      const ext = path.extname(filePath);
      const mime = this.getMime(ext);
      res.writeHead(200, {
        'Content-Type': mime,
        'Content-Length': stats.size,
      });

      const stream = fs.createReadStream(filePath);
      stream.pipe(res);
      stream.on('end', () => this.emit('fileServed', filePath));
      stream.on('error', (err) => this.emit('error', err, filePath, res));
    } catch (err) {
      this.emit('error', err, filePath, res);
    }
  }

  getMime(ext) {
    const map = { '.html': 'text/html', '.css': 'text/css', '.js': 'application/javascript' };
    return map[ext] || 'application/octet-stream';
  }
}
```

... (continued on next page)

```
const server = http.createServer((req, res) => {
  const safePath = decodeURIComponent(req.url.split('?')[0] || '/');
  const normalized = path.normalize(safePath).replace(/^(\.\.[\/\\])+/, '');
  const filePath = path.join(__dirname, 'public', normalized);
  fileServer.serve(filePath, res);
});

const fileServer = new FileServer(path.join(__dirname, 'public'));

fileServer.on('fileServed', (p) => console.log(`✅ Served ${p}`));
fileServer.on('error', (err, p, res) => {
  console.error(`❌ Error serving ${p}:`, err.message);
  if (!res.headersSent) {
    const code = err.code === 'ENOENT' ? 404 : 500;
    res.writeHead(code, { 'Content-Type': 'text/plain' });
    res.end(err.code === 'ENOENT' ? '❌ Not found' : '❌ Server error');
  } else {
    res.destroy();
  }
});

server.listen(4000, () => console.log('🚀 EventEmitter server on :4000'));
```

Benefits

- **Separation of concerns** - the server logic emits events; the HTTP layer only forwards the request.
- **Extensibility** - other parts of the app can listen for `'fileServed'` to log analytics, update a cache, or trigger notifications.

> ***Challenge:** Add a `'log'` event that writes each request to a file `access.log` using `fs.appendFile`.*

Key Takeaways

- **Core Modules** (`fs`, `path`, `http`, `events`) are the building blocks of any Node

backend. They expose both **synchronous** and **asynchronous** APIs; always favor the async versions for scalability.

• The **event loop** is a deterministic state machine with well-defined phases (`timers`, `pending callbacks`, `poll`, `check`, `close`). Understanding which phase your code runs in helps you predict execution order and avoid subtle bugs.

• **Microtasks** (`process.nextTick`, Promise callbacks) run **after each phase** but **before the next phase**. Use them for cleanup or to guarantee ordering, but don't abuse them.

• **Asynchronous I/O** works by delegating blocking system calls to libuv's **thread pool** and re-injecting callbacks into the **poll** queue. This is why a single JavaScript thread can handle thousands of concurrent connections.

• **Callbacks → Promises → async/await** are evolutionary steps toward cleaner, more maintainable code. The underlying event-loop behavior stays the same; only the syntax changes.

• **Streaming** (`fs.createReadStream`) is the most memory-efficient way to serve large files because it leverages back-pressure and keeps the event loop responsive.

• **EventEmitter** is the pattern that powers most Node APIs. By emitting and listening to custom events you can decouple modules, add instrumentation, and build extensible systems.

> ***Next Step:** In Module 4 you'll learn about Express, the de-facto web framework that wraps the `http` module, adds routing, middleware, and simplifies many of the patterns you just built manually.*

Quick Reference Cheat-Sheet

Module	Commonly Used Methods	Sync vs Async	Typical Use
`fs`	`readFile`, `writeFile`, `createReadStream`, `stat`	Both	File I/O, logging, static assets

Module	Commonly Used Methods	Sync vs Async	Typical Use
`path`	`join`, `resolve`, `basename`, `extname`, `dirname`	Sync only	Build cross-platform paths
`http`	`createServer`, `request`, `response.end`, `setHeader`	Async (event-driven)	Build APIs, serve pages
`events`	`new EventEmitter()`, `.on`, `.once`, `.emit`	N/A (event pattern)	Custom events, stream handling

Event Loop Phase	When it runs	Typical callbacks
`timers`	After timeout expires	`setTimeout`, `setInterval`
`pending callbacks`	After I/O callbacks that were deferred	Some `fs` callbacks
`poll`	Checks for new I/O events	Most network/file I/O callbacks
`check`	After poll, before close	`setImmediate`
`close callbacks`	When handles close	`socket.on('close')`
Microtasks (`process.nextTick`, Promise)	**Immediately after the current phase**	`await`, `.then`, `process.nextTick`

You're now equipped to write real-world Node.js backend code that leverages the power of the core modules while respecting the event-loop mechanics that keep your server fast and responsive. Happy coding!

Building a Simple HTTP Server

> ***"A concise goal focuses on practical backend development; beginners need examples and hands-on exercises to grasp asynchronous concepts."***

In the previous modules you set up a clean development environment, explored Node's core modules, and peeked under the hood of the event loop. Now it's time to turn that knowledge into something you can ***see*** and ***interact*** with - a working HTTP server that responds to real browser requests without any third-party framework.

In this chapter you will:

- Spin up a server with the built-in `http` module.
- Parse URLs and query strings using the native `url` module.
- Distinguish request **methods** (GET, POST, etc.) and route accordingly.
- Set response **headers** correctly for JSON, HTML, and static assets.
- Serve a tiny JSON API and a static HTML page from the file system (`fs`).
- Reinforce asynchronous thinking by wiring everything together with callbacks and the event loop.

By the end of the chapter you will have a single `server.js` file that can be run with `node server.js` and will behave like a very small web-application. The code is deliberately minimal so you can see exactly what each line does-no magic, no hidden abstractions.

Introduction

When you first opened a terminal and typed `node`, you were interacting with a **REPL** that executed JavaScript line-by-line. In the "Core Modules & the Event Loop" chapter you learned that Node's **non-blocking I/O** model is powered by a single-threaded event loop that dispatches callbacks whenever an asynchronous operation finishes.

An HTTP server is the perfect playground for those concepts because:

- **Network I/O** (incoming TCP connections) is inherently asynchronous.
- Every request triggers a callback (`requestListener`) that runs on the event loop.
- You will need to **read** files from disk (`fs.readFile`) - another async operation.
- You will **write** data back to the client (`res.end`) - also asynchronous.

By the end of this chapter you will see the event loop in action, ***live***, every time a browser or a command-line client hits your server.

Core Concepts

Below is a quick refresher of the building blocks you will be using. If any of these feel fuzzy, feel free to skim the earlier chapters; the code examples later will reinforce each idea.

1. The `http` Module - Request & Response Objects

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

// The callback receives two objects for every incoming request.
http.createServer((req, res) => {
  // req  → IncomingMessage (read-only)
  // res → ServerResponse   (write-only)
}).listen(3000);
```

- **`req` (IncomingMessage)**
- `req.method` - HTTP verb (e.g., `'GET'`, `'POST'`).
- `req.url` - raw URL string (e.g., `'/api/users?limit=10'`).
- `req.headers` - an object containing all request headers.
- **`res` (ServerResponse)**

- `res.writeHead(statusCode, headersObj)` - sets the status line and response headers.
- `res.write(chunk)` - streams data to the client (optional).
- `res.end([data])` - signals that the response is complete; you can optionally pass the final chunk here.

Both objects are **streams**, which means you can pipe data (e.g., a file read stream) directly into the response without buffering the whole payload in memory.

2. Understanding HTTP Methods

Method	Typical Use	Example in Our Server
GET	Retrieve a resource (no body)	Serve HTML page, return JSON list
POST	Create a new resource (body payload)	Accept JSON payload for a new user
PUT	Replace a resource	Not covered here, but similar to POST
DELETE	Remove a resource	Not covered here
HEAD	Same as GET but no body	Useful for quick health-checks

For a beginner's server we will focus on **GET** (the most common) and **POST** (to illustrate reading a request body).

3. URL Parsing with the `url` Module

Node ships with a lightweight URL parser that can split the path from the query string:

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

// Inside the request listener:
const parsedUrl = new URL(req.url, `http://${req.headers.host}`);
const pathname = parsedUrl.pathname;   // '/api/users'
const searchParams = parsedUrl.searchParams; // URLSearchParams instance
```

- `pathname` tells you ***what*** the client is asking for.
- `searchParams` lets you read query values (`searchParams.get('limit')`).

This is far more reliable than manually splitting on `'?'` or `'/'`.

4. Working with Response Headers

Headers tell the client ***how*** to interpret the response body. The most important for us are:

Header	Meaning	Typical Value
Content-Type	MIME type of the payload	`text/html`, `application/json`, `text/css`, `image/png`
Cache-Control	Caching directives	`no-cache`, `max-age=3600`
ETag	Entity tag for conditional requests	`"abc123"`
Access-Control-Allow-Origin	CORS - who may read the response	`*` (anywhere) or a specific domain

You set them with `res.writeHead` or `res.setHeader`:

```
res.setHeader('Content-Type', 'application/json');
res.writeHead(200); // status line only
```

5. Serving Static Files (`fs`)

The `fs` module can read files from disk **asynchronously**:

```
const fs = require('fs');
fs.readFile('public/index.html', 'utf8', (err, data) => {
  if (err) {
    // handle error
  } else {
    res.setHeader('Content-Type', 'text/html');
    res.end(data);
  }
});
```

A more efficient pattern for larger files is **streaming**:

```
const stream = fs.createReadStream('public/logo.png');
res.setHeader('Content-Type', 'image/png');
stream.pipe(res);
```

Streaming keeps memory usage low because data is sent to the client chunk-by-chunk.

6. Sending JSON Data

JSON is the lingua franca of modern web APIs. To send JSON:

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

You **must** set `Content-Type: application/json`; otherwise the browser may treat the response as plain text.

Practical Application

Below we walk through a complete, self-contained server. All code lives in a single file (`server.js`) plus a tiny `public/` folder for static assets. Feel free to copy the snippets into a fresh project directory and run them step-by-step.

1. Project Layout

```
simple-http-server/
│
├── server.js            # Main server code (the focus of this chapter)
└── public/
    ├── index.html       # A static HTML page we'll serve
    └── style.css        # Optional stylesheet (not required)
```

> ***Tip: Keep the `public/` folder outside of the JavaScript source to avoid accidentally exposing server-side code.***

2. "Hello, World!" - The Minimal Server

Create `server.js` with the absolute bare minimum:

```
// server.js
const http = require('http');

const PORT = 3000;

const server = http.createServer((req, res) => {
  // Log every request - handy for debugging
  console.log(`${req.method} ${req.url}`);

  // Simple static response
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello, World!\n');
});

server.listen(PORT, () => {
  console.log(`🚀 Server listening on http://localhost:${PORT}`);
});
```

Run it:

```
node server.js
```

Open a browser to `http://localhost:3000`. You should see **Hello, World!**. Every request is logged to the console, showing the event-loop in action.

> ***What you just saw***
>
> **** The OS accepted the TCP connection → Node's `http` module emitted a `'request'` event → Your callback ran on the event loop → `res.end` flushed the data back to the client.***

3. Adding a JSON API Endpoint

Let's extend the server to respond with JSON when the client asks for `/api/greeting`.

```
// server.js (continued)
const { URL } = require('url');

const server = http.createServer((req, res) => {
  console.log(`${req.method} ${req.url}`);

  // Parse the URL once for every request
  const parsedUrl = new URL(req.url, `http://${req.headers.host}`);
  const pathname = parsedUrl.pathname;

  // Route 1 - JSON API
  if (pathname === '/api/greeting' && req.method === 'GET') {
    const payload = {
      greeting: 'Hello from the Node.js API!',
      timestamp: new Date().toISOString(),
    };
    const json = JSON.stringify(payload);
    res.setHeader('Content-Type', 'application/json');
    res.writeHead(200);
    return res.end(json);
  }

  // Fallback - 404 Not Found
  res.writeHead(404, { 'Content-Type': 'text/plain' });
  res.end('Not Found\n');
});
```

Test with `curl`:

```
curl http://localhost:3000/api/greeting
```

You should receive:

```
{
  "greeting":"Hello from the Node.js API!",
  "timestamp":"2026-02-14T12:34:56.789Z"
}
```

4. Serving a Static HTML Page

Create `public/index.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Simple HTTP Server</title>
  <link rel="stylesheet" href="/style.css">
</head>
<body>
  <h1>Welcome to My Node.js Server</h1>
  <p>This page is served directly from the file system.</p>
  <pre id="api-data"></pre>

  <script>
    // Fetch the JSON API and display it on the page
    fetch('/api/greeting')
      .then(r => r.json())
      .then(data => {
        document.getElementById('api-data').textContent = JSON.stringify(data, null, 2);
      })
      .catch(err => console.error(err));
  </script>
</body>
</html>
```

(Optional) add `public/style.css` just to show that static assets can be served too:

```
body { font-family: Arial, sans-serif; margin: 2rem; }
h1 { color: #2c3e50; }
pre { background:#f5f5f5; padding:1rem; }
```

Now modify `server.js` to read files from `public/`:

```
// server.js (continued)
const fs = require('fs');
const path = require('path');

// Helper: send a static file
function serveStaticFile(res, filePath, contentType) {
  // Resolve the absolute path to protect against directory traversal
  const absolutePath = path.resolve(__dirname, filePath);

  // Stream the file - efficient for large assets
  const readStream = fs.createReadStream(absolutePath);
  readStream.on('open', () => {
    res.setHeader('Content-Type', contentType);
    // 200 OK by default
    res.writeHead(200);
    readStream.pipe(res);
  });

  // Error handling (e.g., file not found)
  readStream.on('error', err => {
    console.error(`Error reading ${absolutePath}:`, err);
    res.writeHead(500, { 'Content-Type': 'text/plain' });
    res.end('Internal Server Error');
  });
}

// Inside the request listener, after the JSON route:
if (pathname === '/' && req.method === 'GET') {
  // Serve index.html
  return serveStaticFile(res, './public/index.html', 'text/html');
}

// Serve CSS (or any other static asset) based on extension
if (pathname.endsWith('.css') && req.method === 'GET') {
  const cssPath = `./public${pathname}`; // e.g., '/style.css'
  return serveStaticFile(res, cssPath, 'text/css');
}
```

Now reload `http://localhost:3000/` in the browser. You should see the HTML page, styled by CSS, and the JavaScript will fetch the `/api/greeting` JSON and render it inside `<pre>`.

5. A Tiny Router for Multiple Routes & Methods

Hard-coding each `if` statement works for a few endpoints, but as your app grows

you'll want a **router** - a simple object that maps `[method][path]` to a handler function.

```
// server.js (router section)
const router = {
  // GET routes
  'GET /': (req, res) => serveStaticFile(res, './public/index.html', 'text/html'),
  'GET /api/greeting': (req, res) => {
    const payload = { greeting: 'Hello again!', timestamp: new Date().toISOString() };
    res.setHeader('Content-Type', 'application/json');
    res.writeHead(200);
    res.end(JSON.stringify(payload));
  },
  'GET /style.css': (req, res) => serveStaticFile(res, './public/style.css', 'text/css'),

  // POST route demo (echoes back JSON body)
  'POST /api/echo': (req, res) => {
    let body = '';
    req.on('data', chunk => {
      body += chunk;
      // Guard against huge payloads (simple DoS protection)
      if (body.length > 1e6) { // ~1 MB
        req.connection.destroy();
      }
    });
    req.on('end', () => {
      try {
        const parsed = JSON.parse(body);
        res.setHeader('Content-Type', 'application/json');
        res.writeHead(200);
        res.end(JSON.stringify({ youSent: parsed }));
      } catch (e) {
        res.writeHead(400, { 'Content-Type': 'application/json' });
        res.end(JSON.stringify({ error: 'Invalid JSON' }));
      }
    });
  },
};

// Main request listener - the router in action
const server = http.createServer((req, res) => {
  console.log(`${req.method} ${req.url}`);
```

... (continued on next page)

```
  const parsedUrl = new URL(req.url, `http://${req.headers.host}`);
  const key = `${req.method} ${parsedUrl.pathname}`;

  const handler = router[key];
  if (handler) {
    // Found a matching route
    return handler(req, res);
  }

  // No route matched → 404
  res.writeHead(404, { 'Content-Type': 'text/plain' });
  res.end('Not Found\n');
});
```

What changed?

- The router object is a simple key/value map.
- Each key is a string like `'GET /api/greeting'`.
- The value is a **handler function** that receives `(req, res)`.
- The request listener builds the same key and looks it up.

This pattern mirrors what Express does internally, but we built it from scratch using only core modules.

6. Graceful Error Handling

Even with a tiny router you should anticipate errors:

Situation	Recommended Handling
File not found (`fs.readFile` error)	Respond with `404` and a friendly HTML page.
JSON parse error (client sends malformed body)	Respond with `400 Bad Request` and a JSON error object.
Unexpected exception (e.g., typo in a handler)	Wrap each handler in a `try/catch` and return `500 Internal Server Error`.
Uncaught promise rejection	Use `process.on('unhandledRejection', ...)` to

Situation	Recommended Handling
	log and optionally shut down.

Add a tiny helper to wrap handlers:

```
function safeHandler(fn) {
  return (req, res) => {
    try {
      fn(req, res);
    } catch (err) {
      console.error('Handler threw an error:', err);
      res.writeHead(500, { 'Content-Type': 'application/json' });
      res.end(JSON.stringify({ error: 'Internal Server Error' }));
    }
  };
}

// Apply when constructing the router:
router['GET /'] = safeHandler((req, res) => serveStaticFile(res, './public/index.html', 'text/html'));
```

7. Testing with `curl` and a Browser

Command	Expected Result
`curl -i http://localhost:3000/`	`200 OK`, HTML content, `Content-Type: text/html`
`curl -i http://localhost:3000/api/greeting`	`200 OK`, JSON payload
`curl -i -X POST -H "Content-Type: application/json" -d '{"msg":"hi"}' http://localhost:3000/api/echo`	`200 OK`, JSON echo `{ "youSent": { "msg": "hi" } }`
`curl -i http://localhost:3000/does-not-exist`	`404 Not Found`
Open `http://localhost:3000/` in Chrome/Firefox	Page renders, CSS applied, fetches JSON automatically

Tip: Use `-v` (verbose) or `--trace-ascii` with `curl` to see the raw HTTP exchange; this is an excellent way to observe the event-loop in action (you'll see the server logs interleaved with the client request).

8. Exercise - Extend the Server Yourself

Now that you have a functional skeleton, try one (or more) of the following challenges. Write the code, run the server, and verify it works with a browser or `curl`.

1. **Static Assets Folder** - Serve any file inside `public/` automatically (e.g., images, fonts).

Hint: Use `path.extname` to infer MIME type; create a small lookup object (`{ '.png': 'image/png', '.js': 'application/javascript' }`).

2. **Query-String Filtering** - Extend `/api/greeting` to accept a `?name=YourName` query and personalize the greeting (`"Hello, YourName!"`).
3. **Simple In-Memory Store** - Add a `POST /api/users` endpoint that accepts `{ "name": "...", "age": 30 }` and stores it in an array. Provide a `GET /api/users` that returns the array as JSON.
4. **Graceful Shutdown** - Listen for `SIGINT` (Ctrl-C) and close the server with `server.close()`. Log a message when the process exits.
5. **Logging Middleware** - Write a reusable function that logs the method, path, status code, and response time for each request.

When you finish each task, commit the changes to a local Git repo. This mirrors the real-world workflow of iteratively improving a codebase.

Key Takeaways

- **Node's `http` module is all you need** to create a functional web server; no external dependencies are required.
- **Request (`req`) and response (`res`) objects are streams.** Use `res.writeHead`, `res.setHeader`, `res.write`, and `res.end` to control the HTTP response.
- **Parsing URLs** with the built-in `URL` class gives you a clean `pathname` and a `URLSearchParams` object for query strings.

- **Response headers dictate how the client interprets data** (`Content-Type`, `Cache-Control`, etc.). Always set `Content-Type` correctly for JSON (`application/json`) and HTML (`text/html`).
- **Serving files** via `fs.readFile` (small files) or `fs.createReadStream` (large assets) keeps I/O non-blocking and leverages the event loop.
- **A minimal router** can be built as a plain JavaScript object that maps `"METHOD path"` strings to handler functions-this pattern is the foundation of popular frameworks like Express.
- **Error handling** (404, 400, 500) should be explicit; wrap handlers in `try/catch` or a helper (`safeHandler`) to avoid crashing the server.
- **Asynchronous flow**: every network or filesystem operation hands control back to the event loop, allowing Node to serve many concurrent clients with a single thread.
- **Hands-on practice** (curl, browser, extending the code) reinforces the mental model of "request arrives → event loop → handler → response sent → back to event loop".

Congratulations! You have just built a ***real*** HTTP server from first principles. In the next module we'll explore **routing at scale** and introduce a lightweight third-party library that abstracts the patterns you just mastered, allowing you to focus on business logic while still understanding what happens under the hood. Happy coding!

Routing and Express.js Basics

Module 5 of 9 - Learning Node.js for Server-Side Web Applications

> ***Goal:*** *By the end of this chapter you will have a working Express application that can respond to GET, POST, PUT, and DELETE requests, capture data from route parameters and query strings, and understand how routing fits into the larger Node.js ecosystem you explored in the earlier modules.*

Table of Contents

1. [Why Express? Recap of the Built-in HTTP module]
2. [Installing & Initialising an Express Project]
3. [The Anatomy of an Express Application]
4. [Defining Route Handlers]
- 4.1 GET
- 4.2 POST
- 4.3 PUT
- 4.4 DELETE
5. [Capturing Client Input]
- 5.1 Route Parameters (`:id`)
- 5.2 Query Strings (`?search=...`)
- 5.3 Request Body (`req.body`) - using middleware
6. [Putting It All Together - A Mini-API for a Todo List]
7. [Hands-On Exercises]
8. [Debugging Tips & Common Pitfalls]
9. [Key Takeaways]

Introduction

In **Module 3** you learned about Node's core modules-`http`, `fs`, `path`, and the event loop that powers them. You even built a **raw HTTP server** that responded to a single request. While that exercise showed you the power of low-level APIs, real-world back-ends rarely stay at that level.

Enter **Express.js**: a minimal, unopinionated framework that sits on top of the Node `http` module, handling the tedious boilerplate (parsing URLs, headers, request

bodies, etc.) so you can focus on ***what*** your server should do, not ***how*** to do it.

> ***Think of Express as the "router" that sits at the front door of your Node app.***

> ***It decides, "If the client asks for `/users/42` with a GET, hand it to this function; if they POST to `/users`, hand it to that function."***

In this chapter we will:

- Install Express and set up a fresh Node project.
- Learn the **core concepts** of Express routing (methods, paths, middleware).
- Write **practical route handlers** for the four main HTTP verbs.
- Capture data from the URL itself (route parameters), from the query string, and from the request body.

All code snippets are meant to be **copy-and-paste ready**. Feel free to experiment as you read- coding while you learn cements the concepts far better than passive reading.

Core Concepts

1. Express is a thin wrapper around Node's `http` server

When you call `express()`, you get a **function** that is itself a request listener compatible with `http.createServer`. Internally Express does:

```
const http = require('http');
const app  = express();              // app is a function (req, res) => {}
http.createServer(app).listen(3000);
```

So you still have the event-driven nature of Node-every request is an event that

triggers the listener. Express just adds a **routing layer** on top.

2. The Request-Response Cycle

Stage	What Happens?	Express Piece
Incoming TCP connection	Node's `http` server receives the raw bytes.	`http.createServer`
Parsing HTTP headers	Node parses the request line and headers into a `IncomingMessage` object.	`http` core
Routing	Express matches `req.method` + `req.path` against a table of route definitions.	**Router**
Middleware chain	Zero or more functions run sequentially, each optionally calling `next()`.	**Middleware**
Route handler	The final function that sends a response (`res.send`, `res.json`, `res.end`).	**Handler**
Response sent	Node writes the response bytes back to the client and closes the connection (or keeps it alive).	`http` core

Understanding that **routing** is just a step in the cycle helps you debug later-if a request never reaches your handler, it probably got lost in the router or a middleware didn't call `next()`.

3. HTTP Verbs & RESTful Thinking

Verb	Typical Use	Example
GET	Retrieve data (safe, idempotent)	`GET /books` → list all books

Verb	Typical Use	Example
POST	Create a new resource (non-idempotent)	`POST /books` → add a new book
PUT	Replace a resource (idempotent)	`PUT /books/5` → replace book 5
PATCH	Partial update (often used instead of PUT)	`PATCH /books/5` → change title
DELETE	Remove a resource (idempotent)	`DELETE /books/5` → delete book 5

We will focus on the first four verbs, which are enough to illustrate a **CRUD** (Create-Read-Update-Delete) API.

4. Route Paths - Static vs. Dynamic

- **Static path** - exact string match: `app.get('/about', ...)`.
- **Dynamic (parameterized) path** - placeholders start with `:`: `app.get('/users/:id', ...)`.
- The placeholder value ends up on `req.params`.
- **Wildcard (catch-all)** - `*` or `/**` - useful for 404 handling.

5. Query Strings

A URL can carry a **query string** after a `?`. Example:

```
GET /search?term=node&limit=10
```

Express parses this automatically and makes it available on `req.query` as a plain JavaScript object:

```
// req.query === { term: 'node', limit: '10' }
```

6. Body Parsing - Middleware

By default, Express does **not** parse the request body. You need middleware like:

- `express.json()` - parses JSON payloads (`Content-Type: application/json`).
- `express.urlencoded({ extended: true })` - parses URL-encoded form data (`application/x-www-form-urlencoded`).

These middleware functions read the raw request stream, buffer it, and attach the parsed object to `req.body`.

Practical Application

Now let's roll up our sleeves and build a tiny **Todo API** that demonstrates every concept introduced above.

1. Project Setup

Open a terminal and run the following commands (feel free to replace `todo-api` with any name you like):

```
# 1️⃣ Create a folder and initialise npm
mkdir todo-api
cd todo-api
npm init -y          # creates a package.json with default values

# 2️⃣ Install Express (latest stable version)
npm install express

# 3️⃣ (Optional but handy) Install nodemon for auto-restarting during dev
npm install --save-dev nodemon
```

Add a **dev script** to `package.json` so you can start the server with live reload:

```
{
  "name": "todo-api",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js"
  },
  "dependencies": {
    "express": "^4.19.2"
  },
  "devDependencies": {
    "nodemon": "^3.1.0"
  }
}
```

> ***Tip:** Run `npm run dev` to start the server; nodemon watches for file changes and restarts automatically.*

2. Boilerplate - `index.js`

Create `index.js` in the project root:

```
// index.js
const express = require('express');

// 1️⃣ Create the Express application
const app = express();

// 2️⃣ Middleware to parse JSON bodies (important for POST/PUT)
app.use(express.json());

// 3️⃣ Simple health-check route
app.get('/', (req, res) => {
  res.send('🚀 Todo API is up and running!');
});

// 4️⃣ Start listening - 3000 is a conventional dev port
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`🚀 Server listening on http://localhost:${PORT}`);
});
```

Run it:

```
npm run dev
```

Open a browser (or `curl http://localhost:3000`) and you should see "**Todo API is up and running!**".

3. In-Memory Data Store

For learning purposes we'll keep Todo items in an array. In a real project you'd connect to a database (MongoDB, PostgreSQL, ...) but the array illustrates the CRUD flow clearly.

Add the following just **below the `app.use(express.json())`** line:

```
// Temporary in-memory store - will be lost when the server restarts
let todos = [
  { id: 1, title: 'Learn Express', completed: false },
  { id: 2, title: 'Build a REST API', completed: false },
];
let nextId = 3; // simple counter for generating new IDs
```

4. GET - List & Retrieve

4.1 GET `/todos` - return all items

```
// GET /todos  → returns an array of all todos
app.get('/todos', (req, res) => {
  // Optionally support a query string ?completed=true
  const { completed } = req.query; // string or undefined

  // Filter if query param is present
  let result = todos;
  if (completed !== undefined) {
    const want = completed === 'true';
    result = todos.filter(t => t.completed === want);
  }

  res.json(result); // automatically sets Content-Type: application/json
});
```

4.2 GET `/todos/:id` - retrieve a single Todo

```
// GET /todos/:id  → returns the todo with that ID or 404
app.get('/todos/:id', (req, res) => {
  const id = parseInt(req.params.id, 10); // route param is a string
  const todo = todos.find(t => t.id === id);

  if (!todo) {
    // 404 Not Found
    return res.status(404).json({ error: `Todo with id ${id} not found` });
  }

  res.json(todo);
});
```

5. POST - Create a New Todo

Clients will send JSON like `{ "title": "Buy milk" }`. We'll default `completed` to `false`.

```
// POST /todos  → creates a new todo
app.post('/todos', (req, res) => {
  const { title } = req.body; // body already parsed by express.json()

  // Basic validation - title must be a non-empty string
  if (!title || typeof title !== 'string' || title.trim() === '') {
    return res.status(400).json({ error: 'Title is required and must be a non-empty string' });
  }

  const newTodo = {
    id: nextId++,
    title: title.trim(),
    completed: false,
  };

  todos.push(newTodo);
  // 201 Created + Location header pointing to the new resource
  res.status(201)
     .location(`/todos/${newTodo.id}`)
     .json(newTodo);
});
```

6. PUT - Replace an Existing Todo

PUT expects the ***entire*** resource representation. If a field is missing we'll treat it as a client error (400).

```
// PUT /todos/:id  → replace the todo entirely
app.put('/todos/:id', (req, res) => {
  const id = parseInt(req.params.id, 10);
  const index = todos.findIndex(t => t.id === id);

  if (index === -1) {
    return res.status(404).json({ error: `Todo with id ${id} not found` });
  }

  const { title, completed } = req.body;

  // Validate both fields
  if (typeof title !== 'string' || title.trim() === '' || typeof completed !== 'boolean') {
    return res.status(400).json({ error: 'Both title (string) and completed (boolean) are required' });
  }

  const updated = {
    id,
    title: title.trim(),
    completed,
  };

  todos[index] = updated;
  res.json(updated);
});
```

7. DELETE - Remove a Todo

```
// DELETE /todos/:id  → delete the todo, respond 204 No Content
app.delete('/todos/:id', (req, res) => {
  const id = parseInt(req.params.id, 10);
  const index = todos.findIndex(t => t.id === id);

  if (index === -1) {
    return res.status(404).json({ error: `Todo with id ${id} not found` });
  }

  // Remove from array
  todos.splice(index, 1);
  // 204 No Content - no body returned
  res.sendStatus(204);
});
```

8. Full `index.js` (for reference)

```
// ------------------------------------------------------------
//  Todo API - Express.js basics demonstration
// ------------------------------------------------------------
const express = require('express');
const app = express();

// Parse JSON bodies (required for POST/PUT)
app.use(express.json());

// ------------------------------------------------------------
//  In-memory store (will not survive server restarts)
// ------------------------------------------------------------
let todos = [
  { id: 1, title: 'Learn Express', completed: false },
  { id: 2, title: 'Build a REST API', completed: false },
];
let nextId = 3;

// ------------------------------------------------------------
//  Routes
// ------------------------------------------------------------

// Health-check
app.get('/', (req, res) => {
  res.send('✅ Todo API is up and running!');
});

// GET all (optionally filter by completed)
app.get('/todos', (req, res) => {
  const { completed } = req.query;
  let result = todos;
  if (completed !== undefined) {
    const want = completed === 'true';
    result = todos.filter(t => t.completed === want);
  }
  res.json(result);
});

// GET a single todo by ID
app.get('/todos/:id', (req, res) => {
```

... (continued on next page)

```
  const id = parseInt(req.params.id, 10);
  const todo = todos.find(t => t.id === id);
  if (!todo) {
    return res.status(404).json({ error: `Todo with id ${id} not found` });
  }
  res.json(todo);
});

// POST a new todo
app.post('/todos', (req, res) => {
  const { title } = req.body;
  if (!title || typeof title !== 'string' || title.trim() === '') {
    return res.status(400).json({ error: 'Title is required and must be a non-empty string' });
  }
  const newTodo = { id: nextId++, title: title.trim(), completed: false };
  todos.push(newTodo);
  res.status(201).location(`/todos/${newTodo.id}`).json(newTodo);
});

// PUT (replace) an existing todo
app.put('/todos/:id', (req, res) => {
  const id = parseInt(req.params.id, 10);
  const idx = todos.findIndex(t => t.id === id);
  if (idx === -1) {
    return res.status(404).json({ error: `Todo with id ${id} not found` });
  }
  const { title, completed } = req.body;
  if (typeof title !== 'string' || title.trim() === '' || typeof completed !== 'boolean') {
    return res.status(400).json({ error: 'Both title (string) and completed (boolean) are required' });
  }
  const updated = { id, title: title.trim(), completed };
  todos[idx] = updated;
  res.json(updated);
});

// DELETE a todo
app.delete('/todos/:id', (req, res) => {
  const id = parseInt(req.params.id, 10);
  const idx = todos.findIndex(t => t.id === id);
  if (idx === -1) {
```

... (continued on next page)

```
    return res.status(404).json({ error: `Todo with id ${id} not found` });
  }
  todos.splice(idx, 1);
  res.sendStatus(204);
});

// ------------------------------------------------------------
//  Start server
// ------------------------------------------------------------
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`🚀 Server listening on http://localhost:${PORT}`);
});
```

You now have a **fully functional CRUD API** built with Express, using all four HTTP verbs, route parameters, query strings, and request bodies.

Hands-On Exercises

Below are incremental tasks that reinforce each concept. **Do not skip the "run & test" step** - seeing the response in your terminal or via a tool like Postman solidifies learning.

Exercise 1 - Verify GET-All with Query Filtering

1. Start the server (`npm run dev`).
2. In a new terminal, run:

```
curl "http://localhost:3000/todos?completed=true"
```

3. Expect an empty array (`[]`) because no todo is completed yet.
4. Mark the first todo as completed using the **PUT** route (see Exercise 2)and repeat the query. You should now see that todo in the response.

Exercise 2 - Update a Todo with PUT

```
curl -X PUT "http://localhost:3000/todos/1" \
  -H "Content-Type: application/json" \
  -d '{"title":"Learn Express (updated)","completed":true}'
```

- Verify the response contains the updated object.
- Then `GET /todos/1` to confirm persistence in the in-memory array.

Exercise 3 - Create a Todo with Missing Title

```
curl -X POST "http://localhost:3000/todos" \
  -H "Content-Type: application/json" \
  -d '{}'
```

- The server should reply with **400 Bad Request** and a JSON error message.
- This demonstrates **input validation**-a habit you'll keep in production code.

Exercise 4 - Delete a Todo and Check 404

```
curl -X DELETE "http://localhost:3000/todos/2" -i
```

- The response header should show `HTTP/1.1 204 No Content`.
- Now request the same ID again: `curl http://localhost:3000/todos/2`. You should receive **404**.

Exercise 5 - Add a New Middleware (Logging)

1. Insert the following **before** any route definitions (right after `app.use(express.json())`):

```
app.use((req, res, next) => {
  const now = new Date().toISOString();
  console.log(`[${now}] ${req.method} ${req.originalUrl}`);
  next(); // important! otherwise the request hangs
});
```

2. Restart the server and make any request. Observe the console logging each request with timestamp, verb, and path.

This shows how middleware can be used for cross-cutting concerns such as logging, authentication, or error handling.

Exercise 6 - Refactor Routes into a Separate Router

Create a new file `routes/todos.js`:

```
// routes/todos.js
const express = require('express');
const router = express.Router();

// --- copy the route handlers from index.js (GET, POST, etc.) ---
// Remember to export the router at the end:
module.exports = router;
```

Then in `index.js` replace the inline routes with:

```
const todoRouter = require('./routes/todos');
app.use('/todos', todoRouter);
```

Notice how the base path `/todos` is now defined once, and the router only contains the relative ***paths (`/`, `/:id`, etc.).***

Exercise 7 - Bonus: Implement a Simple 404 Handler

Add this **after** all route definitions, but **before** `app.listen`:

```
app.use((req, res) => {
  res.status(404).json({ error: 'Resource not found' });
});
```

Now any request that doesn't match a defined route (e.g., `GET /unknown`) will return a JSON 404.

Debugging Tips & Common Pitfalls

Symptom	Likely Cause	Fix
Server crashes with "Cannot read property 'json' of undefined"	A route handler forgot to include `res` as a parameter or used `res` after an early `return` without sending a response.	Ensure every code path either `res.send`, `res.json`, `res.end`, or `next(err)`.
POST request returns empty `req.body`	Missing `express.json()` middleware or incorrect `Content-Type` header.	Add `app.use(express.json())` and set `Content-Type: application/json` in the client.
GET `/todos/abc` returns 404 even though an item exists	Route param is a string; you compared it to a number without conversion.	Use `parseInt(req.params.id, 10)` or compare as strings.
Route order matters - a later route never fires.	A "catch-all" route (`app.use('*', ...)`) placed before specific routes.	Place generic middleware **after** specific routes.
CORS errors when testing from a browser	No CORS headers.	Install `cors` (`npm i cors`) and add `app.use(require('cors')())`.
Server does not restart after code change	Using plain `node index.js` instead of `nodemon`.	Run `npm run dev` or install a file-watcher.

Quick Debugging Checklist

1. **Console.log** the incoming `req.method`, `req.path`, `req.params`, `req.query`, and `req.body`.
2. Verify **status codes**: 200 (OK), 201 (Created), 204 (No Content), 400 (Bad Request), 404 (Not Found).
3. Use **Postman** or **Insomnia** for a UI-driven way to craft requests and inspect responses.
4. Check the **Node console** for stack traces - they usually point to the exact line where an error originated.

Key Takeaways

- **Express** is a thin, opinion-free layer on top of Node's native `http` server that provides routing, middleware, and convenience helpers.
- **Routing** matches an HTTP verb + URL pattern to a handler function. You can define static, parameterized, and wildcard routes.
- **Route parameters** (`req.params`), **query strings** (`req.query`), and **request bodies** (`req.body`) are the three primary ways a client can send data to the server.
- **Middleware** (`app.use`) lets you run reusable code (e.g., logging, body parsing, authentication) before your route handlers.
- **CRUD** operations map naturally to HTTP verbs: `GET` → Read, `POST` → Create, `PUT` → Replace/Update, `DELETE` → Delete.
- **Validation** and proper **status codes** are essential for a robust API; always respond with a meaningful error when the client sends bad data.
- **Modularisation** (splitting routes into separate routers) keeps your codebase maintainable as the application grows.
- The **event-driven nature** of Node persists under Express-each incoming request triggers the routing chain, and the server stays non-blocking as long as you avoid long-running synchronous code.

With these fundamentals, you're ready to move on to **Module 6**, where we'll explore **middleware in depth**, integrate a **real database**, and start building a **full-stack Todo application** that talks to a front-end UI. Happy coding!

Middleware, Error Handling, and Static Files

Module 6 of 9 - Learning Node.js for Server-Side Web Applications

> ***Goal:** By the end of this chapter you will understand the middleware pattern, know how the request-response cycle works inside Express, be able to use the built-in middleware `express.json()` and `express.static()`, write your own reusable middleware, and implement a clean, centralized error-handling strategy that works for both synchronous and asynchronous routes.*

Table of Contents

1. [Why Middleware Matters](#why-middleware-matters)
2. [The Request-Response Cycle in Express](#request-response-cycle)
3. [Built-In Middleware: `express.json()` and `express.static()`](#built-in-middleware)
4. [Creating Custom Middleware](#custom-middleware)
5. [Error-Handling Basics](#error-handling-basics)
6. [Centralized Async Error Handling](#centralized-async-error-handling)
7. [Putting It All Together - A Mini-API with Static Front-End](#practical-application)
8. [Hands-On Exercises](#exercises)
9. [Key Takeaways](#key-takeaways)

Introduction

When you built a **simple HTTP server** in Module 4 you learned to listen for a request, read the URL, and send back a response string. In Module 5 you added **routing** with Express, making it easy to map `GET /users/:id` to a handler function.

Now you're ready to move beyond "one function per route" and start **structuring** your application the way professional Node.js back-ends are built. That structure revolves around three concepts:

Concept	What it does	Why you need it
Middleware	Small, reusable functions that sit between the raw request and your final route handler.	Keeps code DRY, lets you add cross-cutting concerns (parsing, logging, auth) without cluttering each route.
Error handling	A consistent way to catch and respond to problems, whether they happen synchronously or inside a promise/`async` function.	Prevents the server from crashing and gives clients useful error messages.
Static file serving	Efficiently deliver files (HTML, CSS, JS, images) directly from the filesystem.	Lets you host a front-end (a single-page app, docs, or assets) without a separate web server.

In this chapter we'll explore each of these ideas, see **how they fit into the request-response cycle**, and finish with a **complete, production-ready example** that you can copy-paste into your own project.

Core Concepts

1. Why Middleware Matters

Middleware is the **glue** that turns a raw HTTP request into something your route handlers can work with. Think of it as a **pipeline**:

```
Incoming request → Middleware #1 → Middleware #2 → ... → Route handler → Middleware #N (error, response) → Outgoing response
```

Each middleware function receives three arguments (four for error middleware, see later):

```
function middleware(req, res, next) { ... }
```

- `req` - the **request object** (augmented as it passes through the pipeline).
- `res` - the **response object** (you can also modify it).
- `next` - a **function** that tells Express, "I'm done, hand the request to the next piece of middleware."

If a middleware **does not call `next()`**, the request **stops** there, and the client will wait forever (or until you manually end the response). This is a common bug for beginners.

Benefits of the middleware pattern

- **Separation of concerns** - parsing JSON, logging, authentication, CORS, etc., each live in their own function.
- **Reusability** - the same middleware can be attached to many routes or even the whole app.
- **Composability** - you can stack middleware in any order, creating powerful processing chains.
- **Testability** - each piece can be unit-tested in isolation.

---

2. The Request-Response Cycle in Express

Below is a **simplified diagram** of what happens when a client hits `GET /api/books/42`:

... (continued on next page)

Key observations:

- **Middleware runs in the order you register it** (`app.use(...)`).
- **Route handlers are just another kind of middleware** that typically end the request by sending a response.
- **Error-handling middleware** is special: it has **four** arguments (`err, req, res, next`). Express only calls it when `next(err)` is invoked or an exception bubbles up from an async handler (see later).

---

3. Built-In Middleware

Express ships with a handful of useful middleware functions that you can enable with `app.use()`. The two most common for beginners are:

A. `express.json()` - Body parsing for JSON

When a client sends a **POST** or **PUT** request with a JSON payload, the raw request body is just a stream of bytes. `express.json()` reads that stream, parses it, and attaches the resulting object to `req.body`.

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

// Parse incoming JSON payloads
app.use(express.json());

// Example route that expects JSON
app.post('/api/users', (req, res) => {
  // req.body is now an object
  const { name, email } = req.body;
  // ... create user ...
  res.status(201).json({ id: 123, name, email });
});
```

> ***Tip: Place `express.json()` near the top of your middleware stack, before any route that needs to read `req.body`.***

B. `express.static()` - Serving static assets

If you have a folder named `public` that contains `index.html`, CSS, client-side JavaScript, images, etc., you can let Express serve those files directly:

```
// All files under ./public become reachable at /static/*
app.use('/static', express.static('public'));
```

Now a request to `GET /static/css/style.css` will automatically read `public/css/style.css` from the filesystem and stream it back with the correct `Content-Type` header.

Why not write your own static server? Because `express.static` handles caching, conditional GET (`If-Modified-Since`), MIME type detection, and range requests-all the little details that make static serving fast and reliable.

---

4. Creating Custom Middleware

Custom middleware is just a function with the signature `(req, res, next)`. Below are three common patterns you'll use early on.

4.1. Simple request logger

```
function requestLogger(req, res, next) {
  const now = new Date().toISOString();
  console.log(`[${now}] ${req.method} ${req.originalUrl}`);
  // Continue to the next middleware/route
  next();
}

// Register globally (runs for every request)
app.use(requestLogger);
```

4.2. Authentication guard (demo version)

```
function requireApiKey(req, res, next) {
  const apiKey = req.header('x-api-key');
  if (!apiKey || apiKey !== process.env.API_KEY) {
    // Throw an error that will be caught by our error handler
    const err = new Error('Invalid or missing API key');
    err.status = 401;
    return next(err);
  }
  // API key is valid - proceed
  next();
}

// Apply only to /api routes
app.use('/api', requireApiKey);
```

4.3. Adding a property to `req` (e.g., a DB client)

```
const { MongoClient } = require('mongodb');
const client = new MongoClient(process.env.MONGO_URI);

async function attachDb(req, res, next) {
  if (!client.isConnected()) await client.connect();
  req.db = client.db('myapp'); // Now every handler can do req.db.collection(...)
  next();
}

// Attach early in the stack
app.use(attachDb);
```

> ***Best practice:** Keep middleware pure - don't send a response unless you must (e.g., authentication failures). Let route handlers be responsible for the final output.*

---

5. Error-Handling Basics

5.1. What makes an error handler different?

An error-handling middleware **has four arguments**. Express distinguishes it from regular middleware by the extra `err` parameter:

```
function errorHandler(err, req, res, next) {
  // Log the error (could be to a file, external service, etc.)
  console.error(err.stack);

  // Choose an appropriate status code
  const status = err.status || 500;

  // Respond with JSON (or HTML for browsers)
  res.status(status).json({
    error: {
      message: err.message,
      // In production you might hide the stack trace
      ...(process.env.NODE_ENV !== 'production' && { stack: err.stack })
    }
  });
}
```

You register **once**, after all other `app.use` and route definitions:

```
app.use(errorHandler);
```

5.2. Throwing or passing errors

- **Synchronous route** - just `throw` an error or call `next(err)`.
- **Asynchronous route** - you **must** either `catch` the promise and call `next(err)`, or use a helper that does it for you (see next section).

Example: Synchronous error

```
app.get('/boom', (req, res) => {
  // This will be caught by the error handler automatically
  throw new Error('Synchronous explosion!');
});
```

Example: Asynchronous error (bad way)

```
app.get('/async-bad', async (req, res) => {
  // If this throws, Express won't see it - the request hangs!
  const user = await User.findById(req.params.id);
  res.json(user);
});
```

---

6. Centralized Async Error Handling

Because most real-world routes perform I/O (`await db.query()`, `await fetch()`, etc.), you need a **reliable pattern** to forward asynchronous errors to the error handler.

6.1. The "wrapper" (higher-order) function

```
// asyncWrapper takes an async fn and returns a function that catches rejections
function asyncWrapper(fn) {
  return function (req, res, next) {
    // fn returns a promise - attach .catch and forward the error
    Promise.resolve(fn(req, res, next)).catch(next);
  };
}
```

Now you can write routes without try/catch boilerplate:

```
app.get(
  '/api/books/:id',
  asyncWrapper(async (req, res) => {
    const book = await req.db.collection('books').findOne({ _id: req.params.id });
    if (!book) {
      const err = new Error('Book not found');
      err.status = 404;
      throw err; // asyncWrapper will catch and forward it
    }
    res.json(book);
  })
);
```

6.2. Using a library (optional)

If you prefer not to write your own wrapper, the community provides `express-async-errors`:

```
npm i express-async-errors
```

```
require('express-async-errors'); // patches Express globally

// Now any async route that throws/rejects automatically goes to the error handler
app.get('/api/users/:id', async (req, res) => {
  const user = await req.db.collection('users').findOne({ _id: req.params.id });
  if (!user) throw Object.assign(new Error('User not found'), { status: 404 });
  res.json(user);
});
```

Both approaches are fine; the wrapper gives you explicit control, while the library reduces boilerplate.

Practical Application

Let's build a **complete mini-API** that:

1. Serves a static front-end (`public/` folder).
2. Provides a JSON CRUD API for a "books" collection.
3. Logs every request.
4. Checks for an API key on `/api/*` routes.
5. Handles errors centrally, including async errors.

Project Layout

```
my-library/
 — .env                # environment variables
 —package.json
 —src/
 —index.js          # entry point - Express app
 —middleware/
 —logger.js
 —apiKeyGuard.js
 asyncWrapper.js
 routes/
 books.js
 public/
 —index.html
 css/
 style.css
```

Step-by-Step Implementation

1 Install dependencies

```
npm init -y
npm i express mongodb dotenv
# (optional) npm i express-async-errors
```

2 Create `.env`

```
PORT=3000
API_KEY=supersecret123
MONGO_URI=mongodb://localhost:27017
```

3 `src/middleware/logger.js`

```
// src/middleware/logger.js
function requestLogger(req, res, next) {
  const now = new Date().toISOString();
  console.log(`[${now}] ${req.method} ${req.originalUrl}`);
  next();
}

module.exports = requestLogger;
```

4️⃣ `src/middleware/apiKeyGuard.js`

```
// src/middleware/apiKeyGuard.js
function requireApiKey(req, res, next) {
  const key = req.header('x-api-key');
  if (!key || key !== process.env.API_KEY) {
    const err = new Error('Invalid or missing API key');
    err.status = 401;
    return next(err);
  }
  next();
}

module.exports = requireApiKey;
```

5️⃣ `src/middleware/asyncWrapper.js`

```
// src/middleware/asyncWrapper.js
function asyncWrapper(fn) {
  return function (req, res, next) {
    Promise.resolve(fn(req, res, next)).catch(next);
  };
}

module.exports = asyncWrapper;
```

6️⃣ `src/routes/books.js` - CRUD routes

```
// src/routes/books.js
const express = require('express');
const asyncWrapper = require('../middleware/asyncWrapper');
const router = express.Router();

// GET /api/books - list all books
router.get(
  '/',
  asyncWrapper(async (req, res) => {
    const books = await req.db
      .collection('books')
      .find({})
      .toArray();
    res.json(books);
  })
);

// GET /api/books/:id - single book
router.get(
  '/:id',
  asyncWrapper(async (req, res, next) => {
    const { id } = req.params;
    const book = await req.db.collection('books').findOne({ _id: id });
    if (!book) {
      const err = new Error('Book not found');
      err.status = 404;
      return next(err);
    }
    res.json(book);
  })
);

// POST /api/books - create new book
router.post(
  '/',
  asyncWrapper(async (req, res) => {
    const { title, author } = req.body;
    if (!title || !author) {
      const err = new Error('Missing title or author');
      err.status = 400;
```

... (continued on next page)

```
      throw err;
    }
    const result = await req.db.collection('books').insertOne({ title, author });
    res.status(201).json({ _id: result.insertedId, title, author });
  })
);

// PUT /api/books/:id - update book
router.put(
  '/:id',
  asyncWrapper(async (req, res, next) => {
    const { id } = req.params;
    const { title, author } = req.body;
    const update = { $set: { title, author } };
    const result = await req.db.collection('books').updateOne({ _id: id }, update);
    if (result.matchedCount === 0) {
      const err = new Error('Book not found');
      err.status = 404;
      return next(err);
    }
    res.json({ _id: id, title, author });
  })
);

// DELETE /api/books/:id - delete book
router.delete(
  '/:id',
  asyncWrapper(async (req, res, next) => {
    const { id } = req.params;
    const result = await req.db.collection('books').deleteOne({ _id: id });
    if (result.deletedCount === 0) {
      const err = new Error('Book not found');
      err.status = 404;
      return next(err);
    }
    res.status(204).end(); // No content
  })
);

module.exports = router;
```

> ***Note: MongoDB's default `_id` is an `ObjectId`. For brevity we treat it as a plain string; in a real app you'd `new ObjectId(id)` and handle conversion errors.***

7⃣ `src/index.js` - Assemble everything

```
// src/index.js
require('dotenv').config();          // Load .env
const express = require('express');
const { MongoClient, ObjectId } = require('mongodb');
const path = require('path');

const requestLogger = require('./middleware/logger');
const requireApiKey = require('./middleware/apiKeyGuard');
const booksRouter = require('./routes/books');

const app = express();
const PORT = process.env.PORT || 3000;

// ---------- Global Middleware ----------
app.use(requestLogger);               // 1⃣ Log every request
app.use(express.json());              // 2⃣ Parse JSON bodies

// ---------- Attach DB client ----------
let dbClient; // will hold the connected MongoClient

async function attachDb(req, res, next) {
  try {
    if (!dbClient) {
      dbClient = new MongoClient(process.env.MONGO_URI, { useUnifiedTopology: true });
      await dbClient.connect();
      console.log('✅ Connected to MongoDB');
    }
    // Make the DB accessible as req.db
    req.db = dbClient.db('mylibrary');
    next();
  } catch (err) {
    next(err);
  }
}
app.use(attachDb);

// ---------- Static Files ----------
app.use('/static', express.static(path.join(__dirname, '..', 'public')));

// ---------- API Routes ----------
```

... (continued on next page)

```
app.use('/api', requireApiKey, booksRouter); // API key guard applied to all /api/*

// ---------- Fallback for unknown routes ----------
app.use((req, res, next) => {
  const err = new Error('Not Found');
  err.status = 404;
  next(err);
});

// ---------- Centralized Error Handler ----------
app.use((err, req, res, next) => {
  console.error(`[ERROR] ${err.message}`);
  const status = err.status || 500;
  res.status(status).json({
    error: {
      message: err.message,
      // Hide stack traces in production
      ...(process.env.NODE_ENV !== 'production' && { stack: err.stack })
    }
  });
});

// ---------- Start the Server ----------
app.listen(PORT, () => {
  console.log(`🚀 Server listening on http://localhost:${PORT}`);
});
```

8️⃣ `public/index.html` - A tiny front-end

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>My Library</title>
  <link rel="stylesheet" href="/static/css/style.css">
</head>
<body>
  <h1>My Library</h1>
  <ul id="book-list"></ul>

  <script>
    // Simple client that fetches books and renders them
    async function loadBooks() {
      const resp = await fetch('/api/books', {
        headers: { 'x-api-key': 'supersecret123' }
      });
      const books = await resp.json();
      const list = document.getElementById('book-list');
      list.innerHTML = '';
      books.forEach(b => {
        const li = document.createElement('li');
        li.textContent = `${b.title} - ${b.author}`;
        list.appendChild(li);
      });
    }

    loadBooks().catch(console.error);
  </script>
</body>
</html>
```

9️⃣ Test the whole thing

```
node src/index.js
```

- Open `http://localhost:3000/static/index.html` → you should see the list of books (empty at first).
- Use **Postman**, **cURL**, or the browser console to `POST /api/books` with JSON `{ "title":"Node.js Design Patterns", "author":"Mario Casciaro" }` (remember the `x-api-key` header).

- Reload the page → the new book appears.

You now have a **fully functional** Express app that illustrates:

- **Built-in middleware** (`express.json`, `express.static`).
- **Custom middleware** for logging, API-key guarding, and DB attachment.
- **Centralized, async-aware error handling** that never leaves a request hanging.

Hands-On Exercises

> *Goal: Reinforce each concept with a small, focused task.*

> *Tip: Clone the `my-library` folder, create a new branch for each exercise, and commit your changes.*

#	Exercise	What you'll learn
1	**Add CORS support** - Install the `cors` package and configure it to allow requests only from `http://localhost:3000`.	Using third-party middleware, order of middleware matters.
2	**Create a rate-limiter** - Write middleware that tracks the number of requests per IP per minute and returns **429 Too Many Requests** when the limit is exceeded.	Stateful custom middleware, `res.setHeader`.
3	**Move the async wrapper to a global patch** - Instead of wrapping each route, monkey-patch `express.Router` so that any async handler is automatically wrapped.	Advanced pattern, modifying Express prototypes (understand trade-offs).
4	**Serve a 404 HTML page** - When a static asset is missing, return a custom `404.html` from	Mixing static and dynamic error handling.

#	Exercise	What you'll learn
	the `public/` folder instead of JSON.	
5	**Write unit tests** - Use `jest` and `supertest` to verify that: (a) a request without an API key returns 401, (b) a malformed JSON body returns 400, (c) the logger prints the correct method/path.	Testing middleware in isolation.

Bonus Challenge: Replace the MongoDB client with an **in-memory array** (no external DB) and keep the same API. This will help you see how the `req.db` abstraction decouples route logic from the storage implementation.

Key Takeaways

• **Middleware is the backbone of an Express app** - it lets you compose small, reusable functions that shape every request before it reaches your business logic.

• **Built-in middleware** (`express.json`, `express.static`) handles the most common needs out of the box, saving you from reinventing the wheel.

• **Custom middleware** can do anything: logging, security checks, attaching resources, or even modifying the response. Keep them **pure** (don't end the response unless necessary) and always call `next()` (or `next(err)`).

• **Error handling** in Express is centralized via a function with **four arguments**. By delegating all errors to this handler you guarantee a consistent API and prevent crashes.

• **Async errors** require special attention. Either wrap every async route with a helper (`asyncWrapper`) or use a library like `express-async-errors`.

• **The request-response cycle** follows the order you register middleware. Understanding that order is crucial for things like authentication (must run **before** protected routes) and static serving (often best placed **early**).

- **Static file serving** (`express.static`) is fast, handles caching automatically, and integrates seamlessly with the rest of your middleware stack.

With these patterns under your belt, you can now:

1. **Structure larger applications** - split routes, services, and middleware into separate files/folders.
2. **Add cross-cutting concerns** (security, monitoring, compression) without touching your core business logic.
3. **Maintain a clean error surface** that works for both sync and async code, making debugging and client-side handling far easier.

> ***Next step: In Module 7you'll explore databases and models, learning how to integrate ORMs (like Sequelize or Mongoose) and how to keep your data layer cleanly separated from your routing layer.***

Happy coding!

Working with Databases - MongoDB & Mongoose

Module 7of 9- Learning Node.js for Server-Side Web Applications

> ***Goal: By the end of this chapter you will be able to spin-up a MongoDB database (local or Atlas), model your data with Mongoose schemas and models, and perform full-stack CRUD operations while handling validation errors in an Express app.***

Table of Contents

1. [Introduction](#introduction)
2. [Core Concepts](#core-concepts)

- 2.1 MongoDB fundamentals
- 2.2 The Mongoose library
- 2.3 Schemas, models & validation
- 2.4 Relationships & population
- 2.5 Mongoose middleware (hooks)

3. [Practical Application](#practical-application)
- 3.1 Setting up the environment (local MongoDB vs Atlas)
- 3.2 Project scaffolding - revisiting Express basics
- 3.3 Defining a **Todo** data model
- 3.4 Connecting to the database (centralised connection file)
- 3.5 Implementing CRUD routes with async/await
- 3.6 Handling validation & runtime errors
- 3.7 Bonus: a simple "User ⇔ Todo" relationship
- 3.8 Hands-on exercises for the reader

4. [Key Takeaways](#key-takeaways)

Introduction

When you built the routing and middleware layers in Modules 5 and 6, your server could accept HTTP requests, parse bodies, and return JSON. The missing piece for any real-world API is **persistence** - a place to store data between requests.

MongoDB is a **document-oriented NoSQL database** that stores data as JSON-like ***BSON*** objects. Its flexible schema is a natural match for JavaScript, and the **Mongoose** ODM (Object-Document Mapper) gives you a familiar, model-centric API on top of MongoDB.

In this chapter you will:

- Install and run a MongoDB instance locally **or** connect to a free Atlas cluster.
- Learn the essential Mongoose concepts: **Schema → Model → Document**.

- Write **CRUD** (Create, Read, Update, Delete) endpoints that interact with the database.
- Validate incoming data, surface helpful error messages, and keep your routes tidy using async/await and Express middleware.

By the time you finish, you'll have a fully functional "Todo" API that you can extend, test, and eventually plug into a front-end framework.

Core Concepts

2.1 MongoDB Fundamentals

Concept	Description	Example
Database	A logical container for collections.	`todoApp`
Collection	Analogous to a SQL table; holds documents of the same "type".	`todos`
Document	A single record stored as BSON (binary JSON).	`{ _id: ObjectId(...), title: "Buy milk", completed: false }`
ObjectId	12-byte unique identifier generated by MongoDB.	`507f1f77bcf86cd799439011`
BSON Types	Supports more than JSON (e.g., Date, Binary, Decimal128).	`new Date()` stored as BSON Date

Why MongoDB for beginners?

- No rigid schema - you can evolve your data model without costly migrations.
- Native JSON-like format → less mental translation between client and server.
- Rich query language (filter, projection, aggregation) that scales from simple

to complex.

2.2 The Mongoose Library

Mongoose sits between your Express code and MongoDB. It provides:

- **Schema definitions** - declare the shape, types, defaults, and validation rules of a document.
- **Models** - constructors that let you create, query, update, and delete documents.
- **Middleware (hooks)** - functions that run before/after certain operations (e.g., hashing a password before saving).
- **Population** - a way to "join" related documents across collections.

Install it with:

```
npm i mongoose
```

> ***Tip: Keep the Mongoose version aligned with your Node version. As of this writing, `^7.5.0` works with Node 18+.***

2.3 Schemas, Models & Validation

1 Schema - describes the ***blueprint*** of a document.

```
const { Schema } = require('mongoose');

const todoSchema = new Schema({
  title: {
    type: String,
    required: [true, 'Title is required'],
    minlength: [3, 'Title must be at least 3 characters'],
    maxlength: [100, 'Title cannot exceed 100 characters'],
    trim: true
  },
  description: {
    type: String,
    maxlength: [500, 'Description cannot exceed 500 characters'],
    default: ''
  },
  completed: {
    type: Boolean,
    default: false
  },
  // Example of a custom validator
  priority: {
    type: Number,
    enum: {
      values: [1, 2, 3],
      message: 'Priority must be 1 (low), 2 (medium) or 3 (high)'
    },
    default: 2
  },
  // Timestamps are added automatically if you enable them (see later)
});
```

2️⃣ Model - the compiled version of a schema. It behaves like a class.

```
const mongoose = require('mongoose');
const Todo = mongoose.model('Todo', todoSchema);
```

You can now do:

```
const newTodo = new Todo({ title: 'Learn Mongoose' });
await newTodo.save();          // INSERT
const all = await Todo.find(); // SELECT *
```

3⃣ Validation

Mongoose runs **synchronous** validation rules (e.g., `required`, `enum`) before sending the operation to MongoDB. If validation fails, a `ValidationError` is thrown, containing a `errors` object for each path.

```
try {
  await Todo.create({ title: '' }); // fails minlength
} catch (err) {
  if (err.name === 'ValidationError') {
    console.log(err.errors.title.message); // "Title must be at least 3 characters"
  }
}
```

4⃣ Timestamps - add `createdAt` and `updatedAt` automatically.

```
const todoSchema = new Schema({ ... }, { timestamps: true });
```

2.4 Relationships & Population

MongoDB is ***schema-less***, but most applications need references between documents. The classic pattern is to store an **ObjectId** that points to another collection.

```
// userSchema.js
const userSchema = new Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true }
});
const User = mongoose.model('User', userSchema);

// todoSchema.js (add a reference)
const todoSchema = new Schema({
  // ... previous fields ...
  owner: { type: Schema.Types.ObjectId, ref: 'User', required: true }
});
```

Population pulls the referenced document(s) into the result:

```
const todo = await Todo.findById(id).populate('owner', 'name email');
// `owner` now contains the full user object (only name & email fields)
```

Population is lazy - it adds an extra query under the hood, so use it when you actually need the related data.

2.5 Mongoose Middleware (Hooks)

Hooks let you run code **before** or **after** certain lifecycle events:

Hook type	When it runs	Example use case
`pre('save')`	Before a document is saved (insert or update)	Hash a password, set a slug
`post('save')`	After a document is saved	Log activity, send a webhook
`pre('findOneAndUpdate')`	Before `findOneAndUpdate` executes	Enforce immutable fields
`post('remove')`	After a document is removed	Clean up related files or cache

Simple pre-save hook (auto-generate a URL-friendly slug):

```
todoSchema.pre('save', function (next) {
  if (this.isModified('title')) {
    this.slug = this.title
      .toLowerCase()
      .replace(/[^\w]+/g, '-')
      .replace(/^-+|-+$/g, '');
  }
  next();
});
```

> ***Best practice:** Keep hooks short and non-blocking. If you need async work (e.g., hashing), return a promise or call `next(err)`.*

Practical Application

We will now build a **Todo API** from scratch, wiring everything together. The final folder structure looks like this:

```
todo-api/
├── node_modules/
├── src/
│   ├── config/
│   │   └── db.js              # DB connection logic
│   ├── models/
│   │   ├── Todo.js
│   │   └── User.js
│   ├── routes/
│   │   └── todos.js
│   ├── middleware/
│   │   └── errorHandler.js
│   ├── app.js                 # Express app
│   └── server.js              # Entry point
├── .env
├── package.json
└── README.md
```

3.1 Setting Up the Environment

3.1.1 Option A - Run MongoDB Locally

1. **Install MongoDB Community Server** (Windows/macOS/Linux).

- macOS (Homebrew): `brew tap mongodb/brew && brew install mongodb-community@6.0`
- Windows: use the MSI installer from the official site.

2. **Start the daemon**

```
# macOS & Linux
mongod --dbpath /usr/local/var/mongodb

# Windows (default service)
net start MongoDB
```

The server listens on `mongodb://127.0.0.1:27017` by default.

3.1.2 Option B - Use MongoDB Atlas (cloud)

1. Create a free Atlas account at https://cloud.mongodb.com.
2. Build a **cluster** (choose "Shared Cluster - Free").
3. In **Network Access** → ***IP Whitelist*** add `0.0.0.0/0` (or your IP).
4. In **Database Access** → ***Create a Database User*** (username/password).
5. Copy the **connection string** (looks like `mongodb+srv://<user>:<pwd>@cluster0.mongodb.net/<dbname>?retryWrites=true&w=majority`).

> ***Tip: Store the connection string in a `.env` file (never commit it).***

```
# .env
MONGODB_URI=mongodb://127.0.0.1:27017/todoApp   # local
# or
# MONGODB_URI=mongodb+srv://user:pwd@cluster0.mongodb.net/todoApp?retryWrites=true&w=majority
PORT=3000
```

3.2 Project Scaffolding - Revisiting Express Basics

Initialize the project (you already have Express from earlier modules):

```
mkdir todo-api && cd todo-api
npm init -y
npm i express mongoose dotenv morgan cors
npm i -D nodemon
```

Add a **dev script** to `package.json`:

```
"scripts": {
  "start": "node src/server.js",
  "dev": "nodemon src/server.js"
}
```

Create the entry point `src/server.js`:

```
require('dotenv').config();               // Load .env
const app = require('./app');              // Express app
const connectDB = require('./config/db');  // DB connection

// Connect to MongoDB first
connectDB()
  .then(() => {
    const PORT = process.env.PORT || 3000;
    app.listen(PORT, () => console.log(`🚀 Server running on http://localhost:${PORT}`));
  })
  .catch(err => {
    console.error('❌ Failed to connect to DB', err);
    process.exit(1);
  });
```

And `src/app.js` (the core Express configuration):

```
const express = require('express');
const morgan = require('morgan');
const cors = require('cors');
const todoRoutes = require('./routes/todos');
const errorHandler = require('./middleware/errorHandler');

const app = express();

// Middleware stack - same pattern you learned in Module 6
app.use(cors());                // Enable CORS for all origins (dev only)
app.use(morgan('dev'));         // HTTP request logger
app.use(express.json());        // Parse JSON bodies

// Routes
app.use('/api/todos', todoRoutes);

// 404 handler for unknown routes
app.use((req, res, next) => {
  const err = new Error(` Not Found - ${req.originalUrl}`);
  err.status = 404;
  next(err);
});

// Centralised error handling (see later)
app.use(errorHandler);

module.exports = app;
```

3.3 Defining a Todo Data Model

Create `src/models/Todo.js`:

```
const mongoose = require('mongoose');
const { Schema } = mongoose;

// 1️⃣ Schema definition
const todoSchema = new Schema(
  {
    title: {
      type: String,
      required: [true, 'Title is required'],
      minlength: [3, 'Title must be at least 3 characters'],
      maxlength: [100, 'Title cannot exceed 100 characters'],
      trim: true
    },
    description: {
      type: String,
      maxlength: [500, 'Description cannot exceed 500 characters'],
      default: ''
    },
    completed: {
      type: Boolean,
      default: false
    },
    priority: {
      type: Number,
      enum: {
        values: [1, 2, 3],
        message: 'Priority must be 1 (low), 2 (medium) or 3 (high)'
      },
      default: 2
    },
    // Optional reference to a user (later chapter)
    owner: {
      type: Schema.Types.ObjectId,
      ref: 'User',
      required: false
    }
  },
  {
    timestamps: true, // adds createdAt & updatedAt
    versionKey: false // remove __v field
```

... (continued on next page)

```
  }
);

// 2️⃣ Pre-save hook - generate a slug (nice for URLs)
todoSchema.pre('save', function (next) {
  if (this.isModified('title')) {
    this.slug = this.title
      .toLowerCase()
      .replace(/[^\w]+/g, '-')
      .replace(/^-+|-+$/g, '');
  }
  next();
});

// 3️⃣ Model compilation
module.exports = mongoose.model('Todo', todoSchema);
```

> ***Note: The `owner` field is optional now; we'll add a User model later in the bonus section.***

3.4 Connecting to the Database (Centralised Connection File)

Create `src/config/db.js`:

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

const MONGODB_URI = process.env.MONGODB_URI;

/**
 * Connects to MongoDB using Mongoose.
 * Returns a Promise that resolves when the connection is ready.
 */
async function connectDB() {
  if (!MONGODB_URI) {
    throw new Error('❌ MONGODB_URI is not defined in .env');
  }

  // Set some useful options (Mongoose 7+ defaults are already sensible)
  const options = {
    // useNewUrlParser: true, // no longer needed
    // useUnifiedTopology: true,
  };

  await mongoose.connect(MONGODB_URI, options);
  console.log('✅ Connected to MongoDB');
}

// Export the function so server.js can await it
module.exports = connectDB;
```

3.5 Implementing CRUD Routes with async/await

Create `src/routes/todos.js`:

```
const express = require('express');
const router = express.Router();
const Todo = require('../models/Todo');

// ------------------------------------------------
// Helper - async wrapper to avoid try/catch everywhere
// ------------------------------------------------
const asyncHandler = fn => (req, res, next) => {
  Promise.resolve(fn(req, res, next)).catch(next);
};

// -------------------
// CREATE - POST /api/todos
// -------------------
router.post(
  '/',
  asyncHandler(async (req, res) => {
    // Mongoose will validate automatically based on the schema
    const todo = await Todo.create(req.body);
    res.status(201).json({ success: true, data: todo });
  })
);

// -------------------
// READ ALL - GET /api/todos
// -------------------
router.get(
  '/',
  asyncHandler(async (req, res) => {
    // Optional query parameters: ?completed=true&priority=1
    const filter = {};
    if (req.query.completed) filter.completed = req.query.completed === 'true';
    if (req.query.priority) filter.priority = Number(req.query.priority);

    const todos = await Todo.find(filter).sort({ createdAt: -1 });
    res.json({ success: true, count: todos.length, data: todos });
  })
);

// -------------------
```

... (continued on next page)

```
// READ ONE - GET /api/todos/:id
// -------------------
router.get(
  '/:id',
  asyncHandler(async (req, res, next) => {
    const todo = await Todo.findById(req.params.id);
    if (!todo) {
      // Let the error handler turn this into a 404 response
      const err = new Error('Todo not found');
      err.status = 404;
      return next(err);
    }
    res.json({ success: true, data: todo });
  })
);

// -------------------
// UPDATE - PUT /api/todos/:id
// -------------------
router.put(
  '/:id',
  asyncHandler(async (req, res, next) => {
    const options = {
      new: true,            // return the updated doc
      runValidators: true   // enforce schema validation on update
    };
    const todo = await Todo.findByIdAndUpdate(req.params.id, req.body, options);
    if (!todo) {
      const err = new Error('Todo not found');
      err.status = 404;
      return next(err);
    }
    res.json({ success: true, data: todo });
  })
);

// -------------------
// DELETE - DELETE /api/todos/:id
// -------------------
router.delete(
```

... (continued on next page)

```
  '/:id',
  asyncHandler(async (req, res, next) => {
    const todo = await Todo.findByIdAndDelete(req.params.id);
    if (!todo) {
      const err = new Error('Todo not found');
      err.status = 404;
      return next(err);
    }
    res.status(204).end(); // No Content
  })
);

module.exports = router;
```

Why `asyncHandler`?

Every route returns a promise (thanks to `await`). If an error is thrown, Express won't catch it automatically; we must forward it to `next(err)`. The tiny wrapper does that for us, keeping the route bodies clean and focused on business logic.

3.6 Handling Validation & Runtime Errors

Create a central error-handling middleware: `src/middleware/errorHandler.js`

```
/**
 * Centralised error handling middleware.
 * It captures both synchronous and asynchronous errors,
 * formats them, and sends a JSON response.
 */
function errorHandler(err, req, res, next) {
  console.error(err); // Log for debugging (could be replaced by winston)

  // Default values
  let status = err.status || 500;
  let message = err.message || 'Internal Server Error';

  // Mongoose validation errors have a special shape
  if (err.name === 'ValidationError') {
    status = 400;
    const errors = Object.values(err.errors).map(e => e.message);
    message = 'Validation failed';
    return res.status(status).json({
      success: false,
      message,
      errors,
      // useful during development, hide in production
      stack: process.env.NODE_ENV === 'development' ? err.stack : undefined
    });
  }

  // CastError - invalid ObjectId format
  if (err.name === 'CastError' && err.kind === 'ObjectId') {
    status = 400;
    message = 'Invalid ID format';
  }

  // Duplicate key error (e.g., unique email)
  if (err.code && err.code === 11000) {
    status = 409;
    const field = Object.keys(err.keyValue)[0];
    message = `${field} already exists`;
  }

  res.status(status).json({
```

... (continued on next page)

```
    success: false,
    message,
    // Expose stack only in dev
    stack: process.env.NODE_ENV === 'development' ? err.stack : undefined
  });
}

module.exports = errorHandler;
```

How it works in practice:

```
POST /api/todos
{
  "title": "Hi"
}
```

Response:

```
{
  "success": false,
  "message": "Validation failed",
  "errors": [
    "Title must be at least 3 characters"
  ]
}
```

All other route handlers (`next(err)`) will be caught by this middleware, ensuring a consistent JSON error format across the API.

3.7 Bonus: A Simple "User ↔ Todo" Relationship

Let's quickly add a **User** model and protect the Todo routes so that each Todo belongs to a user. This demonstrates `populate`, reference fields, and a tiny authentication stub (no JWT yet - just a static user for the demo).

3.7.1 User Model (`src/models/User.js`)

```
const mongoose = require('mongoose');
const { Schema } = mongoose;

const userSchema = new Schema(
  {
    name: { type: String, required: true, trim: true },
    email: {
      type: String,
      required: true,
      unique: true,
      lowercase: true,
      trim: true,
      match: [/^\S+@\S+\.\S+$/, 'Invalid email address']
    },
    // In a real app you'd store a hashed password
    password: { type: String, required: true, minlength: 6 }
  },
  { timestamps: true }
);

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

3.7.2 Dummy Authentication Middleware

Create `src/middleware/fakeAuth.js`. It pretends that every request is made by the **first user** in the database.

```
const User = require('../models/User');

/**
 * Attaches `req.user` with a User document.
 * In production you'd verify a JWT or session cookie.
 */
async function fakeAuth(req, res, next) {
  // For simplicity, find any user; create one if none exists
  let user = await User.findOne();
  if (!user) {
    user = await User.create({
      name: 'Demo User',
      email: 'demo@example.com',
      password: 'secret123' // never store plain text in real apps!
    });
  }
  req.user = user; // expose downstream
  next();
}

module.exports = fakeAuth;
```

Add it to the Express stack **before** the Todo routes (in `src/app.js`):

```
const fakeAuth = require('./middleware/fakeAuth');
app.use(fakeAuth); // <-- inserted after cors & json parser
```

3.7.3 Update the Todo Schema (owner is required)

```
owner: {
  type: Schema.Types.ObjectId,
  ref: 'User',
  required: true
}
```

3.7.4 Adjust the CRUD routes

- When **creating** a Todo, inject `owner: req.user._id`.

- When **reading** Todos, filter by `owner`.
- Use `.populate('owner', 'name email')` to return user data.

Modified snippets (only the changed parts):

```
// CREATE
router.post(
  '/',
  asyncHandler(async (req, res) => {
    const payload = { ...req.body, owner: req.user._id };
    const todo = await Todo.create(payload);
    res.status(201).json({ success: true, data: todo });
  })
);

// READ ALL (only own todos)
router.get(
  '/',
  asyncHandler(async (req, res) => {
    const filter = { owner: req.user._id };
    // optional query filters preserved...
    const todos = await Todo.find(filter)
      .populate('owner', 'name email')
      .sort({ createdAt: -1 });
    res.json({ success: true, count: todos.length, data: todos });
  })
);

// READ ONE - also enforce ownership
router.get(
  '/:id',
  asyncHandler(async (req, res, next) => {
    const todo = await Todo.findOne({ _id: req.params.id, owner: req.user._id })
      .populate('owner', 'name email');
    if (!todo) {
      const err = new Error('Todo not found or not owned by you');
      err.status = 404;
      return next(err);
    }
    res.json({ success: true, data: todo });
  })
);

// UPDATE & DELETE - same ownership check
router.put(
```

... (continued on next page)

```
  '/:id',
  asyncHandler(async (req, res, next) => {
    const todo = await Todo.findOneAndUpdate(
      { _id: req.params.id, owner: req.user._id },
      req.body,
      { new: true, runValidators: true }
    );
    if (!todo) {
      const err = new Error('Todo not found or not owned by you');
      err.status = 404;
      return next(err);
    }
    res.json({ success: true, data: todo });
  })
);

router.delete(
  '/:id',
  asyncHandler(async (req, res, next) => {
    const todo = await Todo.findOneAndDelete({ _id: req.params.id, owner: req.user._id });
    if (!todo) {
      const err = new Error('Todo not found or not owned by you');
      err.status = 404;
      return next(err);
    }
    res.status(204).end();
  })
);
```

Now every Todo belongs to a user, and the API only exposes the authenticated user's data. This pattern scales nicely when you replace `fakeAuth` with real JWT authentication later in the course.

3.8 Hands-On Exercises

Exercise 1 - Switch to Atlas

1. Create a free Atlas cluster.

2. Replace the `MONGODB_URI` in `.env` with the Atlas connection string.
3. Run `npm run dev` and verify that the Todo endpoints still work.

Exercise 2 - Add a "Due Date" Field

- Extend `todoSchema` with a `dueDate: Date` field that **must be in the future**.
- Use a custom validator:

```
dueDate: {
  type: Date,
  validate: {
    validator: v => v > new Date(),
    message: 'Due date must be in the future'
  }
}
```

- Update the POST/PUT routes to accept the new field and test validation errors.

Exercise 3 - Pagination & Sorting

- Modify the `GET /api/todos` route to accept `?page=2&limit=10&sort=priority`.
- Implement safe defaults (`page=1`, `limit=20`).
- Return pagination meta (`totalDocs`, `totalPages`, `currentPage`).

Exercise 4 - Soft Delete (Logical Deletion)

- Add a `deletedAt: Date` field to the schema.
- Change the DELETE endpoint to set `deletedAt` instead of removing the document.
- Ensure the "read all" route filters out soft-deleted docs (`{ deletedAt: null }`).

Exercise 5 - Write a Unit Test with Jest (optional)

- Install `jest supertest`.
- Write a test that posts a new Todo, then fetches it by ID, asserting the response shape and status codes.

Key Takeaways

- **MongoDB** stores data as flexible JSON-like documents, perfect for rapid prototyping.
- **Mongoose** adds a schema layer, model methods, validation, and middleware on top of raw MongoDB.
- **Schema → Model → Document** is the core workflow: define a schema, compile a model, then create/query documents.
- **Async/await + centralized error handling** keep route code readable and ensure consistent JSON error responses.
- **Validation** (required, minlength, enum, custom validators) runs before data hits the database, protecting integrity early.
- **Relationships** are expressed via `ObjectId` references; `populate` lets you "join" related data on demand.
- **Mongoose middleware (hooks)** are powerful for automating repetitive tasks (e.g., slug generation, password hashing).
- **Connection abstraction** (`config/db.js`) lets you swap between a local MongoDB instance and a cloud Atlas cluster by changing a single environment variable.
- **CRUD endpoints** follow a predictable pattern:
- `POST /api/todos` → `Model.create()`
- `GET /api/todos` → `Model.find()` with optional filters
- `GET /api/todos/:id` → `Model.findById()` (or `findOne` for ownership)
- `PUT /api/todos/:id` → `Model.findByIdAndUpdate()` with `runValidators`
- `DELETE /api/todos/:id` → `Model.findByIdAndDelete()` (or soft-delete)

By mastering these fundamentals, you now have a solid backend foundation that can be expanded with authentication, more complex queries, and even GraphQL later in the course. Happy coding!

Authentication & Security

Module 8 of 9 — Learning Node.js for Server-Side Web Applications

> ***Learning Goal:** By the end of this chapter you will be able to hash passwords securely with bcrypt, issue and verify JSON Web Tokens (JWT) for stateless authentication, and apply basic security hardening techniques (Helmet, CORS, rate-limiting) in an Express-based API that stores users in MongoDB with Mongoose.*

> ***Why it matters:** Authentication is the gate-keeper of any web service. A single mistake — storing plain-text passwords, exposing secret keys, or allowing unlimited login attempts — can give attackers full control of your system and your users' data. This chapter shows you how to build a secure, production-ready authentication layer while keeping the code approachable for beginners.*

Introduction

When you built middleware in Module 6 you learned how to **intercept** a request, do something useful (e.g., log, parse JSON), and either pass control forward or end the response. In Module 7 you added a **persistent data layer** with Mongoose, letting you store and retrieve users.

Authentication is the next logical step:

1. **Identify *who*** a client claims to be (login).
2. **Validate** that claim (check password, verify token).
3. **Authorize** the client to access protected resources (middleware that checks the token).

In a modern API we usually go **stateless**-the server does **not** keep a session object in memory or a database. Instead the client receives a **cryptographically signed token** (a JWT) that it sends back on each request. The server can verify the token **without** looking up any session data, which scales beautifully for micro-services, serverless functions, and horizontal scaling.

But security isn't just about tokens. We also need to protect the **transport layer** (HTTPS), **sanitize inputs**, **limit abusive traffic**, and **shield HTTP headers** that leak information. Express gives us a clean way to plug in these defenses as middleware-exactly the pattern you mastered earlier.

> ***Bottom line: By the end of this chapter you'll have a single Express app that demonstrates all the best-practice pieces you need for a real-world login system.***

Core Concepts

Below is a quick-reference map of the concepts we'll weave together. Feel free to skim the definitions first, then come back when the code examples need them.

1. Password Hashing vs. Plain Text

✅ **Good Practice**	❌ **Bad Practice**
Store **hashes** (one-way, irreversible)	Store raw passwords
Use a ***slow*** hashing algorithm (bcrypt, argon2)	Use fast hash (MD5, SHA-1)
Add a **unique salt** per password	Use a fixed salt or no salt

Why? If an attacker steals your database, a plain-text password gives them immediate access-and many users reuse passwords across sites. A strong hash makes it computationally expensive (minutes or hours) to recover the original password,

buying you time to detect and respond to a breach.

2. bcrypt Basics

- **Salt** - Random data mixed into the password before hashing, preventing pre-computed rainbow-table attacks.
- **Cost factor** - Number of rounds (2^cost). Higher cost → slower hash. Typical values: `10-12` for most apps (adjust based on CPU).

Workflow (async/await style):

```
import bcrypt from 'bcrypt';

const saltRounds = 12;               // cost factor
const plain = 'mySuperSecret123';

// 1️⃣ Generate a salt (optional - bcrypt can generate internally)
const salt = await bcrypt.genSalt(saltRounds);

// 2️⃣ Hash the password
const hash = await bcrypt.hash(plain, salt);

// 3️⃣ Verify later
const matches = await bcrypt.compare(plain, hash); // true
```

All bcrypt methods return **Promises**, which makes them perfect to use with `async/await`-the same async pattern you already used for DB queries.

3. JSON Web Tokens (JWT) Anatomy

A JWT is a three-part string separated by dots:

```
header.payload.signature
```

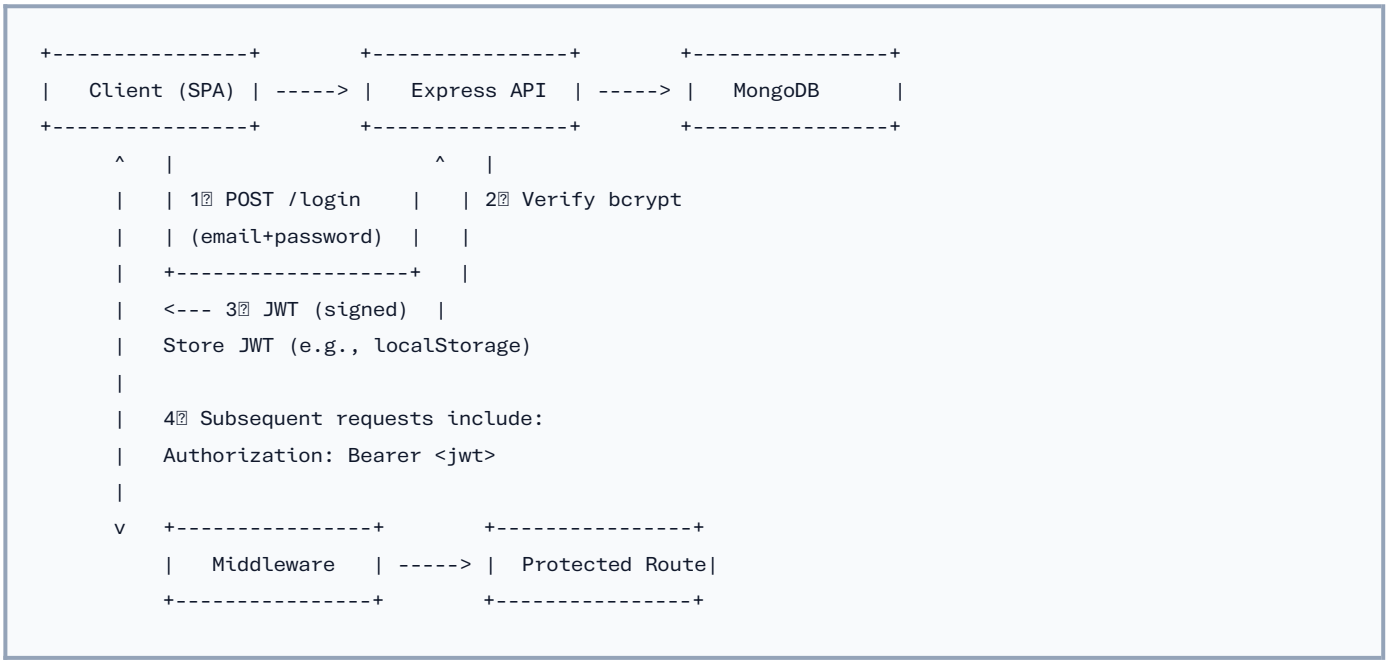
Part	What it contains	Example (Bas
Header	Algorithm (`alg`) & token type (`typ`)	`eyJhbGciOiJIU`

Part	What it contains	Example (Bas
Payload	Claims (user id, role, expiration, custom data)	`eyJ1c2VySWQiO`
Signature	HMAC or RSA signature over `header.payload` using a secret/key	`SflKxwRJSMeKK`

Key points for beginners

- The payload is **not encrypted**-anyone can decode it (Base64URL). Only the signature guarantees integrity.
- Include an **expiration (`exp`)** claim to limit token lifetime.
- Keep the **secret key** out of source code-use environment variables (`process.env.JWT_SECRET`).

4. Stateless Authentication Flow



- **Step 1:**Client sends credentials.
- **Step 2:**Server checks password with bcrypt.
- **Step 3:**If valid, server signs a JWT and returns it.
- **Step 4:**Client attaches JWT to `Authorization` header on every protected

request.

• **Middleware** verifies the token, extracts the user id, and attaches the user object to `req`.

5. Common Attack Vectors (and how we'll mitigate them)

Attack	What it does	Mitigation in this chapter
Brute-force login	Repeated password attempts until success.	Rate-limiting (`express-rate-limit`).
Credential stuffing	Uses leaked credentials from other sites.	Same as brute-force + short token expiry.
Cross-Site Scripting (XSS)	Injects malicious script into responses.	Helmet sets `Content-Security-Policy`, `X-XSS-Protection`.
Cross-Site Request Forgery (CSRF)	Tricks a logged-in browser to send unwanted requests.	Stateless APIs with JWT in `Authorization` header are immune; if you use cookies, add `csurf`.
Man-in-the-Middle (MITM)	Intercepts traffic to read tokens.	Enforce HTTPS (outside Node scope) + short-lived JWTs.
Information leakage via headers	Server reveals tech stack, version numbers.	Helmet hides `X-Powered-By`, sets `X-Content-Type-Options`.

6. Security-Focused Middleware

Middleware	Purpose	Install
helmet	Sets a collection of secure HTTP headers.	`npm i helmet`
cors	Controls which origins may access the API.	`npm i cors`
express-rate-limit	Limits repeated requests from same IP.	`npm i express-rate-limit`
express-mongo-sanitize	Removes `$` and `.` from req	`npm i express-mongo-sanitize`

Middleware	Purpose	Install
(optional)	bodies to prevent NoSQL injection.	

We'll wire them **early** in the request-pipeline so they affect every route.

7. Environment Variables & Secret Management

Never hard-code secrets. Use a `.env` file (with `dotenv`) or a cloud secret manager.

```
# .env (never commit!)
JWT_SECRET=super-strong-random-string-that-is-32+bytes
BCRYPT_SALT_ROUNDS=12
PORT=3000
MONGO_URI=mongodb://localhost:27017/auth-demo
```

Load them at app start:

```
import dotenv from 'dotenv';
dotenv.config();
```

Practical Application

Below is a **complete, runnable example** that ties all concepts together. Feel free to copy the code into a fresh folder and run `npm install` followed by `npm start`. The example assumes you have MongoDB running locally (or you can replace the URI with Atlas).

1. Project Setup

```
# 1️⃣ Create a new folder
mkdir node-auth-demo && cd node-auth-demo

# 2️⃣ Initialise npm
npm init -y

# 3️⃣ Install dependencies
npm i express mongoose bcrypt jsonwebtoken helmet cors express-rate-limit dotenv
# dev dependency for auto-reloading (optional but handy)
npm i -D nodemon
```

Add a `**start**` script to `package.json`:

```
"scripts": {
  "start": "nodemon index.js"
}
```

Create the file structure:

```
node-auth-demo/
│
├─ .env
├─ index.js
├─ models/
│   └─ User.js
├─ routes/
│   └─ auth.js
└─ middleware/
    └─ authMiddleware.js
```

2. `.env` - Keep it secret!

```
# .env
PORT=3000
MONGO_URI=mongodb://localhost:27017/auth-demo
JWT_SECRET=superSecretKeyThatShouldBe32CharsOrMore!
BCRYPT_SALT_ROUNDS=12
```

3. `index.js` - Bootstrap the server

```
// index.js
import express from 'express';
import mongoose from 'mongoose';
import helmet from 'helmet';
import cors from 'cors';
import rateLimit from 'express-rate-limit';
import dotenv from 'dotenv';
import authRoutes from './routes/auth.js';
import { errorHandler, notFound } from './middleware/errorHandler.js'; // from Module 6

dotenv.config();

const app = express();

// ---------- 1️⃣ Global Middleware ----------
app.use(helmet());                      // security headers
app.use(cors({ origin: '*' }));         // allow any origin for demo; tighten in prod
app.use(express.json());                // parse JSON bodies

// ---------- 2️⃣ Rate Limiting ----------
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15️⃣ min
  max: 100,                 // limit each IP to 100 requests per window
  standardHeaders: true,
  legacyHeaders: false,
});
app.use(limiter);           // apply to all routes

// ---------- 3️⃣ Routes ----------
app.use('/api/auth', authRoutes);

// ---------- 4️⃣ 404 & Error handling ----------
app.use(notFound);          // from Module 6
app.use(errorHandler);    // from Module 6

// ---------- 5️⃣ Connect to DB & start ----------
const PORT = process.env.PORT || 3000;
mongoose
  .connect(process.env.MONGO_URI, {
    useNewUrlParser: true,
```

... (continued on next page)

```
    useUnifiedTopology: true,
  })
  .then(() => {
    console.log('✅ Connected to MongoDB');
    app.listen(PORT, () => console.log(`🚀 Server running on http://localhost:${PORT}`));
  })
  .catch((err) => {
    console.error('❌ MongoDB connection error:', err);
    process.exit(1);
  });
```

> *Note: `errorHandler` and `notFound` are the same generic error-handling middleware you built in Module 6. Re-using them demonstrates how authentication errors fit into a unified error pipeline.*

4. `models/User.js` - Mongoose schema with password hashing

```
// models/User.js
import mongoose from 'mongoose';
import bcrypt from 'bcrypt';
import dotenv from 'dotenv';
dotenv.config();

const SALT_ROUNDS = parseInt(process.env.BCRYPT_SALT_ROUNDS) || 12;

const userSchema = new mongoose.Schema(
  {
    email: {
      type: String,
      required: [true, 'Email required'],
      unique: true,
      lowercase: true,
      trim: true,
      match: [/^\S+@\S+\.\S+$/, 'Invalid email format'],
    },
    password: {
      type: String,
      required: [true, 'Password required'],
      minlength: [6, 'Password must be at least 6 chars'],
    },
    name: {
      type: String,
      required: [true, 'Name required'],
      trim: true,
    },
    // optional: role, createdAt, etc.
    role: {
      type: String,
      enum: ['user', 'admin'],
      default: 'user',
    },
  },
  { timestamps: true }
);

/**
 * Pre-save hook: hash password only if it's new or modified.
```

... (continued on next page)

```
 */
userSchema.pre('save', async function (next) {
  // `this` refers to the document being saved
  if (!this.isModified('password')) return next();

  try {
    const salt = await bcrypt.genSalt(SALT_ROUNDS);
    this.password = await bcrypt.hash(this.password, salt);
    next();
  } catch (err) {
    next(err);
  }
});

/**
 * Instance method to compare candidate password with stored hash.
 */
userSchema.methods.isPasswordMatch = async function (candidatePwd) {
  return bcrypt.compare(candidatePwd, this.password);
};

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

> ***Why a pre-save hook?***

It guarantees that **every** time a password field changes (e.g., during registration or a password reset) it gets hashed automatically, eliminating the risk of accidentally saving plain text.

5. `routes/auth.js` - Register, login, and a protected test route

```
// routes/auth.js
import express from 'express';
import jwt from 'jsonwebtoken';
import User from '../models/User.js';
import asyncHandler from 'express-async-handler';
import { protect } from '../middleware/authMiddleware.js';
import dotenv from 'dotenv';
dotenv.config();

const router = express.Router();

// ---------------------------------------------------------------------
// 1️⃣ Register endpoint
// ---------------------------------------------------------------------
router.post(
  '/register',
  asyncHandler(async (req, res) => {
    const { name, email, password } = req.body;

    // Basic validation (you can expand with Joi or express-validator)
    if (!name || !email || !password) {
      res.status(400);
      throw new Error('Please provide name, email and password');
    }

    // Check for existing user
    const exists = await User.findOne({ email });
    if (exists) {
      res.status(409);
      throw new Error('User already exists with that email');
    }

    // Create user - password will be hashed by pre-save hook
    const user = await User.create({ name, email, password });

    // Respond with a JWT (optional: you could require email verification first)
    const token = generateToken(user._id);
    res.status(201).json({
      _id: user._id,
      name: user.name,
```

... (continued on next page)

```
      email: user.email,
      role: user.role,
      token,
    });
  })
);

// ---------------------------------------------------------------------
// 2️⃣ Login endpoint
// ---------------------------------------------------------------------
router.post(
  '/login',
  asyncHandler(async (req, res) => {
    const { email, password } = req.body;

    // Find user by email
    const user = await User.findOne({ email });
    if (!user) {
      res.status(401);
      throw new Error('Invalid email or password');
    }

    // Compare password
    const isMatch = await user.isPasswordMatch(password);
    if (!isMatch) {
      res.status(401);
      throw new Error('Invalid email or password');
    }

    // Success - generate JWT
    const token = generateToken(user._id);
    res.json({
      _id: user._id,
      name: user.name,
      email: user.email,
      role: user.role,
      token,
    });
  })
);
```

... (continued on next page)

```
// ---------------------------------------------------------------------
// 3️⃣ Example protected route
// ---------------------------------------------------------------------
router.get(
  '/profile',
  protect, // <-- auth middleware verifies token first
  asyncHandler(async (req, res) => {
    // `req.user` is attached by the protect middleware
    const user = await User.findById(req.user.id).select('-password');
    if (!user) {
      res.status(404);
      throw new Error('User not found');
    }
    res.json(user);
  })
);

// ---------------------------------------------------------------------
// Helper: create a signed JWT
// ---------------------------------------------------------------------
function generateToken(userId) {
  // Expires in 1 hour - adjust to your needs
  return jwt.sign({ id: userId }, process.env.JWT_SECRET, { expiresIn: '1h' });
}

export default router;
```

Explanation of key pieces

- `asyncHandler` (from `express-async-handler`) wraps async route handlers, automatically forwarding thrown errors to the error-handling middleware-just like you did for other routes in Module 6.
- `protect` middleware (next section) **verifies** the JWT and attaches `req.user`.
- The register route returns a token right away to simplify the demo. In a production app you might **require email verification** before issuing a token.

6. `middleware/authMiddleware.js` - Verify JWT

```
// middleware/authMiddleware.js
import jwt from 'jsonwebtoken';
import asyncHandler from 'express-async-handler';
import User from '../models/User.js';
import dotenv from 'dotenv';
dotenv.config();

/**
 * protect - Express middleware that checks for a valid JWT.
 * If valid, attaches the user payload to req.user and calls next().
 */
export const protect = asyncHandler(async (req, res, next) => {
  let token;

  // Tokens are usually sent in the Authorization header:
  //   Authorization: Bearer <token>
  if (
    req.headers.authorization &&
    req.headers.authorization.startsWith('Bearer')
  ) {
    token = req.headers.authorization.split(' ')[1];
  }

  if (!token) {
    res.status(401);
    throw new Error('Not authorized, token missing');
  }

  try {
    // Verify token and decode payload
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    // Attach the full user object (minus password) to request
    req.user = { id: decoded.id };
    next();
  } catch (err) {
    res.status(401);
    throw new Error('Not authorized, token invalid or expired');
  }
});
```

Why store only the user ID? JWT payload should be **minimal**. The ID lets us fetch fresh data from the database (e.g., updated roles). Storing the entire user object could expose stale or sensitive data if the token is compromised.

7. Testing the Flow

You can use **Postman**, **Insomnia**, or `curl`. Below are quick curl commands.

7.1 Register a new user

```
curl -X POST http://localhost:3000/api/auth/register \
  -H "Content-Type: application/json" \
  -d '{"name":"Alice","email":"alice@example.com","password":"Secret123"}'
```

Expected response (status 201):

```
{
  "_id": "65c5f1b2c2e9b5a9d6e8a4f7",
  "name": "Alice",
  "email": "alice@example.com",
  "role": "user",
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6..."
}
```

Copy the `token` value for the next step.

7.2 Log in (existing user)

```
curl -X POST http://localhost:3000/api/auth/login \
  -H "Content-Type: application/json" \
  -d '{"email":"alice@example.com","password":"Secret123"}'
```

Response similar to register (new token).

7.3 Access a protected route

```
curl http://localhost:3000/api/auth/profile \
  -H "Authorization: Bearer <PASTE_TOKEN_HERE>"
```

Success (status 200):

```
{
  "_id": "65c5f1b2c2e9b5a9d6e8a4f7",
  "name": "Alice",
  "email": "alice@example.com",
  "role": "user",
  "createdAt": "2024-02-14T12:34:56.789Z",
  "updatedAt": "2024-02-14T12:34:56.789Z",
  "__v": 0
}
```

If you omit the token or use an expired one, you'll receive a **401 Unauthorized** error with a clear message-thanks to the `protect` middleware and the global error handler.

8. Adding Rate Limiting to Auth Endpoints Only

Brute-force attacks target login endpoints. We can tighten limits specifically for `/login` and `/register` while keeping a more generous global limit.

```
// routes/auth.js (add near top)
import rateLimit from 'express-rate-limit';

// Apply stricter limits only to auth routes
const authLimiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 min
  max: 5,                   // max 5 attempts per IP
  message:
    'Too many login attempts from this IP, please try again after 15 minutes',
  standardHeaders: true,
  legacyHeaders: false,
});

router.use(authLimiter); // all routes defined after this line will be rate-limited
```

Now a malicious script can only try five password combinations per IP every 15 minutessignificantly raising the cost of a credential-stuffing attack.

9. Hardening with Helmet & CORS

We already added `app.use(helmet())` and `app.use(cors(...))` in `index.js`. Let's briefly explain each header Helmet sets for you:

Header	What it does	Why it matters
`Content-Security-Policy`	Restricts where scripts, styles, images can be loaded from.	Mitigates XSS by blocking inline scripts.
`X-Content-Type-Options: nosniff`	Prevents browsers from MIME-sniffing a response.	Stops an attacker from executing malicious files masquerading as safe types.
`X-Frame-Options: DENY`	Disallows embedding the page in `<iframe>`.	Prevents clickjacking.
`Referrer-Policy: no-referrer`	Controls how much referrer info is sent.	Reduces leakage of URLs that may contain sensitive query strings.
`X-XSS-Protection: 0` (modern browsers)	Turns off old buggy XSS filter.	Modern CSP is more reliable.
`X-Powered-By: Express` (removed)	Hides the fact you're using Express.	Makes fingerprinting harder.

CORS is set to `origin: '*'` for the demo, but in production you'd replace it with a whitelist:

```
app.use(
  cors({
    origin: ['https://myfrontend.com', 'https://admin.myapp.com'],
    credentials: true, // if you ever switch to cookie-based auth
  })
);
```

10. Full-stack Integration (Optional)

If you have a simple React or Vue front-end, you can now:

- POST `/api/auth/register` → store token in **localStorage** or **HttpOnly cookie** (the latter is more secure).
- On every API call, add `Authorization: Bearer <token>` header.
- Use a React context or Vuex store to keep the current user state, refreshing it via `/api/auth/profile`.

This is **outside the scope** of the current chapter, but the patterns are the same- your front-end just needs to respect the stateless token contract.

Key Takeaways

- **Never store plain-text passwords.** Use `bcrypt` (or Argon2) with a per-user salt and a sensible cost factor (`10-12`).
- **JWTs provide stateless authentication.** They contain only non-sensitive claims, are signed with a secret, and should have an expiration (`exp`).
- **Protect routes with middleware.** A `protect` function verifies the token, extracts the user id, and attaches it to `req`.
- **Rate limiting is essential** for login and registration endpoints to thwart brute-force attacks.
- **Helmet, CORS, and other security middlewares** are cheap ways to add HTTP-header hardening and origin restrictions.
- **Environment variables** keep secrets out of source control. Use a `.env` file locally and a secret manager in production.
- **Error handling stays consistent.** Throw an `Error` inside async route handlers; your global error middleware (from Module 6)will format the response.
- **Async/await** works seamlessly with both Mongoose and bcrypt, keeping the code readable and avoiding callback hell.

Further Reading & Tools

Topic	Resource
Password hashing	<https://cheatsheetseries.owasp.org/cheatsheets/Password***Storage***Cheat_Sheet.html>
JWT best practices	<https://jwt.io/introduction/>
Helmet docs	<https://helmetjs.github.io/>
Express-rate-limit	<https://github.com/nfriedly/express-rate-limit>
CORS	<https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>
Node security checklist	<https://github.com/snyk/nodejs-security-checklist>
Testing auth flows	Postman's **Auth** tab + automated tests (see Postman docs).
Static analysis	`npm i -D npm-audit` or `npm i -D snyk` to scan dependencies.

🎉 You did it!

You now have a **complete, production-ready authentication system** built with the same Express fundamentals you learned earlier. The next module (Module 9) will show you how to **deploy** this API securely to the cloud, integrate **refresh tokens**, and add **role-based access control**. Keep this chapter handy—security bugs are often introduced later, and the patterns here will help you spot and fix them quickly. Happy coding!

Deployment, Environment Variables, and Scaling

Module 9 of 9 - Learning Node.js for Server-Side Web Applications

> ***Learning Goal:** By the end of this chapter you will be able to configure environment variables with `dotenv`, deploy a Node.js/Express app to a cloud platform (Heroku, Render, or Railway), use a process manager (PM2) to keep your service alive, and understand the basics of scaling a production-grade API.*

Table of Contents

Section	What you'll learn
Introduction	Why deployment, env-vars, and scaling matter for real-world back-ends.
Core Concepts	• Environment variables & `dotenv` • Preparing an app for production (scripts, `Procfile`, static assets) • Deploying to three popular PaaS providers • Process managers (PM2) • Horizontal vs. vertical scaling, health checks, and zero-downtime restarts
Practical Application	Step-by-step hands-on: turn the **Todo-API** you built in Modules 7-8 into a production-ready service, push it to **Render**, add PM2, and test scaling.
Key Takeaways	Quick reference checklist.

Introduction

You've already built a functional API that talks to MongoDB, validates input, hashes passwords, and issues JWTs. Until now you've been running it locally with

`node index.js` (or `npm run dev`). In the real world the code lives on a server, must survive crashes, and must be reachable by users worldwide.

Deployment is the act of moving your source code from your laptop to a remote machine (or a cluster of machines) that will run it 24/7.

Environment variables are the ***secret sauce*** that lets the same code run in many different contexts- local development, staging, production- without hard- coding credentials or configuration details.

Scaling is the process of making sure your API can handle more traffic, either by giving a single instance more resources (vertical) or by adding more instances behind a load balancer (horizontal).

In this chapter we'll stitch all of those ideas together, ending with a **deploy-and-scale checklist** that you can reuse for any future Node.js project.

Core Concepts

1. Environment Variables & the `dotenv` Library

1.1 Why not hard-code secrets?

Reason	Explanation
Security	If you push a repo with a plain-text password to GitHub, anyone can see it.
Portability	The same code can run on a developer's laptop (`localhost:27017`) and on a cloud DB (`cluster0.mongodb.net`).
Configuration Drift	Production may need a different JWT expiration time, log level, or feature flag.

Reason	Explanation
Compliance	Many standards (PCI-DSS, GDPR) require secrets to be stored outside the code base.

1.2 The `.env` file

Create a file named **`.env`** at the root of your project (same folder as `package.json`). It contains key-value pairs, one per line:

```
# .env - never commit this file to a public repo!
PORT=4000
MONGODB_URI=mongodb+srv://myUser:myPass@cluster0.mongodb.net/todoapp?retryWrites=true&w=majority
JWT_SECRET=superSecretKeyThatShouldBe32+Chars
NODE_ENV=development          # change to "production" on the server
```

> ***Tip:** Add `.env` to your `.gitignore` (most starter projects already do this).*

1.3 Loading variables with `dotenv`

Install the library:

```
npm i dotenv
```

At the very top of **`index.js`** (or wherever you start your app) add:

```
require('dotenv').config(); // loads .env into process.env
```

Now you can read the values anywhere:

```
const PORT = process.env.PORT || 3000;
const DB_URI = process.env.MONGODB_URI;
const JWT_SECRET = process.env.JWT_SECRET;
```

> ***Important:*** `process.env` ***values are always strings. Convert to numbers or booleans when needed:***

```
const isProd = process.env.NODE_ENV === 'production';
const maxConnections = Number(process.env.MAX_CONN) || 10;
```

1.4 Using env-vars in other modules

If you have separate files (e.g., `config/db.js`), you can centralise the logic:

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

module.exports = {
  port: parseInt(process.env.PORT, 10) || 3000,
  mongodbUri: process.env.MONGODB_URI,
  jwtSecret: process.env.JWT_SECRET,
  nodeEnv: process.env.NODE_ENV || 'development',
};
```

Then import:

```
const { mongodbUri } = require('./config/config');
mongoose.connect(mongodbUri, { /* options */ });
```

---

2. Preparing Your App for Production

2.1 Scripts in `package.json`

Script	Purpose
`"start"`	The command the platform will run to start the server (must not use `nodemon`).
`"dev"`	Runs the server with `nodemon` for live reload during development.
`"build"`	If you have a front-end bundle (React, Vue) that needs to be compiled, put the command here.
`"postinstall"`	Runs after dependencies are installed on the host; useful for building assets.

Example `package.json` snippet:

```
{
  "name": "todo-api",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "dev": "nodemon index.js",
    "start": "node index.js",
    "build": "echo 'no front-end to build'",
    "postinstall": "npm run build"
  },
  "dependencies": {
    "express": "^4.18.2",
    "mongoose": "^7.2.0",
    "dotenv": "^16.3.1",
    "jsonwebtoken": "^9.0.0",
    "bcryptjs": "^2.4.3"
  },
  "devDependencies": {
    "nodemon": "^3.0.1"
  }
}
```

2.2 The `Procfile` (Heroku-style)

Many PaaS providers (Heroku, Render, Railway) look for a `**Procfile**` to know how to launch your app. It's a plain-text file with the syntax:

```
web: npm start
```

If you need a background worker (e.g., for a job queue) you could add:

```
worker: node workers/emailWorker.js
```

2.3 Handling Uncaught Errors

A production server should **never crash** because of an unhandled promise rejection or exception. Add a global error handler early in `index.js`:

```
process.on('uncaughtException', (err) => {
  console.error('⚠ Uncaught Exception:', err);
  // optional: send alert to monitoring service, then exit gracefully
  process.exit(1);
});

process.on('unhandledRejection', (reason, promise) => {
  console.error('⚠ Unhandled Rejection at:', promise, 'reason:', reason);
  // same as above - you may want to log and exit
});
```

Later we'll see how PM2 can automatically restart the process when it exits.

2.4 Enabling CORS & Security Headers for Production

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

app.use(cors({
  origin: process.env.CORS_ORIGIN || '*', // restrict in prod
  credentials: true,
}));
app.use(helmet()); // sets sensible HTTP headers
```

2.5 Logging

For a simple start, use `console.log` with timestamps. In real projects you'd switch to `winston` or `pino`. Example wrapper:

```
function log(...args) {
  console.log(new Date().toISOString(), ...args);
}
log('Server started on', PORT);
```

---

3. Deploying to a Cloud Platform

We'll walk through **three** platforms that offer a free tier for hobby projects. Pick whichever you like; the steps are largely interchangeable.

Platform	Why use it?	Typical Free Tier
Heroku	Very beginner-friendly, Git-based deploy, good docs.	550-dyno-hours/month (roughly 1 dyno running continuously).
Render	Automatic HTTPS, free static site hosting, easy background workers.	750 hoursof web service + 750 hoursof background worker per month.
Railway	"Deploy from GitHub" wizard, built-in PostgreSQL & MongoDB plugins.	$5 credit per month (enough for small Node app).

> ***Note: All three platforms automatically set `NODE_ENV=production` for you. They also expose environment variables via their UI, which we'll use instead of a `.env` file on the server.***

3.1 Common Prerequisites

1. **Git repository** - Ensure your code is committed and pushed to a remote (GitHub, GitLab, Bitbucket).
2. **`package.json` with a `start` script** - The platform runs `npm start`.
3. **`Procfile` (optional but recommended)** - Clarifies the process type.
4. **MongoDB Atlas cluster** - You already used this in Module 7,so you just need the connection string.
5. **Add a `.gitignore` entry for `.env`** - Never push secrets.

3.2 Deploying to Heroku

1. **Create a Heroku account** → <https://dashboard.heroku.com>
2. **Install the Heroku CLI**

```
curl https://cli-assets.heroku.com/install.sh | sh
# verify
heroku --version
```

3. **Log in**

```
heroku login
```

4. **Create the app**

```
heroku create todo-api-demo
```

Heroku will assign a random sub-domain like `todo-api-demo.herokuapp.com`.

5. **Add the MongoDB URI and JWT secret as config vars**

```
heroku config:set MONGODB_URI="mongodb+srv://user:pass@cluster0.mongodb.net/todoapp"
heroku config:set JWT_SECRET="superSecretKeyThatShouldBe32+Chars"
```

You can also set `PORT` (Heroku overrides it automatically, so you can just ignore it in code).

6. **Push the code**

```
git push heroku main   # or `git push heroku master` depending on branch name
```

Heroku will:

- Detect a Node.js app (via `package.json`).
- Run `npm install`.
- Run `npm run build` if defined in `postinstall`.
- Start the process defined in `Procfile` (`web: npm start`).

7. **Open the app**

```
heroku open
```

Or visit the URL in your browser.

8. **View logs**

```
heroku logs --tail
```

This is handy for debugging deployment errors (e.g., missing env-var).

Scaling on Heroku (horizontal)

```
# add a second dyno (instance)
heroku ps:scale web=2
```

Heroku will automatically load-balance requests across the two dynos. The free tier only allows **one dyno**, but the command demonstrates the concept.

3.3 Deploying to Render

1. **Sign up** → <https://render.com> and link your GitHub account.
2. **Create a New Web Service**

- **Repository**: select your repo.
- **Branch**: `main` (or whichever you want).
- **Build Command**: `npm install` (or `npm ci`).
- **Start Command**: `npm start`.

3. **Environment Variables**

In the "Environment" tab, add the same keys you used locally (`MONGODB_URI`, `JWT_SECRET`, `PORT` optional). Render injects them into the container.

4. **Automatic Deploy**

Render builds a Docker-like container, runs it, and gives you a URL (e.g., `https://todo-api-demo.onrender.com`).

5. **HTTPS**

Render automatically provisions a free TLS certificate, so you can safely call your API from browsers or mobile apps.

6. **Horizontal Scaling**

- Click **Settings → Autoscaling**.
- Set "Minimum Instances" = 1, "Maximum Instances" = 3 (or any number you're

comfortable with).

• Render will spin up extra containers when CPU or request count crosses thresholds.

3.4 Deploying to Railway

1. **Create an account** → <https://railway.app> and connect to GitHub.
2. **New Project → Deploy from GitHub**

• Choose the repo / branch.

• Railway auto-detects Node.js and creates a "service".

3. **Add Variables**

In the "Variables" tab, add `MONGODB_URI`, `JWT_SECRET`. Railway also injects a `RAILWAY_ENVIRONMENT` variable (`production` by default).

4. **Deploy**

Railway runs the build and start steps automatically. The service's public URL appears in the dashboard.

5. **Background Workers**

If you later add a job queue (e.g., Bull or Agenda), you can spin up a ***worker*** service in Railway with its own start command (`node workers/emailWorker.js`).

6. **Scaling**

Railway's free tier gives you **500 MB RAM** and **1 vCPU** To scale vertically, click "Upgrade" → select a higher-memory plan. Horizontal scaling is done by adding **additional instances** (still limited on free tier).

---

4. Process Managers - Keeping Your App Alive

When you run `node index.js` manually, any crash (e.g., uncaught exception) will stop the server. In production you want **automatic restarts**, **log rotation**, and the

ability to **zero-downtime reloads**. Enter **PM2**.

4.1 Installing PM2

```
npm i -g pm2   # global install (requires sudo on Linux/macOS)
```

4.2 Starting the App

```
pm2 start index.js --name todo-api
```

PM2 creates a **process list** (`pm2 list`) where you can see status, memory, and CPU usage.

4.3 Managing the Process

Command	What it does
`pm2 stop todo-api`	Gracefully stops the process.
`pm2 restart todo-api`	Restarts, re-reading the source (good after a code change).
`pm2 reload todo-api`	Zero-downtime reload (keeps connections alive).
`pm2 delete todo-api`	Removes it from PM2's registry.
`pm2 logs`	Streams stdout/stderr from all PM2-managed apps.
`pm2 save`	Persists the current process list so it can be resurrected after a server reboot.
`pm2 startup`	Generates a systemd (or upstart) script that launches PM2 on boot.

Example:

```
pm2 startup systemd   # prints a command you need to run with sudo
sudo env PATH=$PATH:/usr/local/bin pm2 startup systemd -u $USER --hp $HOME
pm2 save
```

Now, if your VM restarts, PM2 will automatically resurrect `todo-api`.

4.4 Using an ecosystem file

For larger projects you can store configuration in `ecosystem.config.js`:

```
// ecosystem.config.js
module.exports = {
  apps: [
    {
      name: 'todo-api',
      script: './index.js',
      instances: 'max',           // cluster mode (one process per CPU core)
      exec_mode: 'cluster',
      env: {
        NODE_ENV: 'development',
        PORT: 3000,
      },
      env_production: {
        NODE_ENV: 'production',
        PORT: 8080,
      },
    },
  ],
};
```

Start with:

```
pm2 start ecosystem.config.js --env production
```

Cluster mode spawns multiple Node processes that share the same server port (Node's built-in load balancer). This is a simple way to achieve **horizontal scaling on a single machine**.

4.5 Health Checks & Monitoring

PM2 includes a built-in web dashboard:

```
pm2 monit
```

Or you can enable the **PM2 Plus** SaaS monitoring (free tier available). For production you may also integrate with **Grafana**, **Prometheus**, or services like **New Relic**.

---

5. Scaling - From One Instance to Many

5.1 Vertical vs. Horizontal

Scaling type	Definition	Pros	Cons
Vertical	Add more CPU/RAM to a single server (e.g., upgrade from a 1 GB to a 2 GBdroplet).	Simpler, no need for load balancer.	Limited by hardware ceiling; a single point of failure.
Horizontal	Add more ***instances*** (containers, dynos, pods) behind a load balancer.	Near-infinite capacity, fault-tolerant.	Requires orchestration (Kubernetes, Render autoscaling, etc.).

In the context of our free-tier platforms:

- **Heroku** - horizontal scaling via multiple dynos (requires paid plan).
- **Render** - autoscaling horizontally with multiple containers.
- **Railway** - vertical scaling via larger instance size (horizontal limited on free tier).

5.2 Statelessness - The Key to Horizontal Scaling

Your API must be **stateless**: any request can be handled by any instance without relying on in-memory session data.

- **JWT** is perfect because the token carries all the information the server needs.
- **MongoDB** is the single source of truth for data.
- **Cache** (Redis) can be externalized if you later need it.

If you ever need to store temporary data (e.g., file uploads), store them in **S3** or another cloud bucket, not on the local filesystem.

5.3 Load Balancing Basics

- **Platform-provided**: Render, Railway, and Heroku all expose a built-in load balancer that routes traffic round-robin.
- **Custom**: For self-hosted VMs, you can use **NGINX** as a reverse proxy:

```
upstream todo_api {
    server 127.0.0.1:3000;
    server 127.0.0.1:3001;
    server 127.0.0.1:3002;
}

server {
    listen 80;
    location / {
        proxy_pass http://todo_api;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }
}
```

Then run three PM2 instances in **fork mode** (or cluster mode) on ports 3000-3002.

5.4 Zero-Downtime Deploys

When you push a new version, you don't want users to see "Service Unavailable". Strategies:

1. **PM2 `reload`** - gracefully restarts each cluster worker after it finishes current requests.
2. **Blue-Green Deploy** - spin up a new version on a different sub-domain, switch DNS or load balancer once health checks pass.
3. **Canary Releases** - route a small percentage of traffic to the new version, monitor, then roll out fully.

For beginner projects, **PM2 reload** is the simplest and works on any host.

5.5 Monitoring & Autoscaling Signals

Most platforms expose **metrics** (CPU, memory, request latency). You can set thresholds that trigger scaling:

- **Render** - "Scale when CPU > 70% for 2 minutes".
- **Heroku** - Use **Metrics** dashboard; add a third-party add-on like **Hirefire** for auto-scaling.
- **Railway** - Manual scaling via UI; future roadmap includes autoscaling.

Best practice: set alerts (email, Slack) for abnormal spikes so you can intervene before users notice downtime.

Practical Application

Below is a **step-by-step hands-on walkthrough** that ties together everything we've discussed. By the end you'll have a fully-deployed, PM2-managed, horizontally-scalable Todo-API running on Render.

> ***Prerequisites***
>
> ***- Node ≥ 18 installed locally.***
>
> ***- GitHub account with the repository from Modules 7-8 (the Todo-API).***

> *- MongoDB Atlas cluster (you already have one).*

5.1 Refactor the Existing Code for Production

5.1.1 Add `dotenv` and Central Config

```
npm i dotenv
```

Create **`config/config.js`**:

```
// config/config.js
require('dotenv').config(); // loads .env in dev, ignored in prod

module.exports = {
  port: parseInt(process.env.PORT, 10) || 3000,
  mongodbUri: process.env.MONGODB_URI,
  jwtSecret: process.env.JWT_SECRET,
  nodeEnv: process.env.NODE_ENV || 'development',
  corsOrigin: process.env.CORS_ORIGIN || '*',
};
```

Update **`index.js`** (or `app.js`) to use the config:

```
// index.js
const express = require('express');
const mongoose = require('mongoose');
const helmet = require('helmet');
const cors = require('cors');
const { port, mongodbUri, nodeEnv, corsOrigin } = require('./config/config');

const app = express();

app.use(express.json());
app.use(helmet());
app.use(cors({ origin: corsOrigin, credentials: true }));

// ... import routes, error handling, etc.

// Connect to MongoDB
mongoose
  .connect(mongodbUri, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
  })
  .then(() => console.log('✅ Connected to MongoDB'))
  .catch((err) => {
    console.error('❌ MongoDB connection error:', err);
    process.exit(1);
  });

app.listen(port, () => {
  console.log(`🚀 Server listening on http://localhost:${port} (${nodeEnv})`);
});
```

5.1.2 Add a `Procfile`

Create a file named **`Procfile`** (no extension) in the repo root:

```
web: npm start
```

5.1.3 Add a `postinstall` script (optional)

If you ever add a front-end, you'd compile it here. For now we'll keep it simple:

```
"scripts": {
  "dev": "nodemon index.js",
  "start": "node index.js",
  "postinstall": "echo 'postinstall - nothing to build'"
}
```

5.1.4 Add a `.gitignore` entry

```
# .gitignore
node_modules/
.env
```

Commit everything:

```
git add .
git commit -m "Prepare for production deployment"
git push origin main
```

5.2 Create a Render Service

1. **Log in** to Render → **New** → **Web Service**.
2. **Connect** to your GitHub repo, select the `main` branch.
3. **Build Command**: `npm install` (Render automatically runs `npm ci` if a lockfile exists).
4. **Start Command**: `npm start`.

Render will read the `Procfile` and run `npm start` accordingly.

5. **Environment** → **Add Variables**

Variable	Value
`MONGODB_URI`	***Your Atlas connection string***
`JWT_SECRET`	***A strong random string***

Variable	Value
`PORT` (optional)	`10000` (Render sets its own internal port, but you can keep the default).
`CORS_ORIGIN`	`https://your-frontend.render.com` (or `*` for testing).

6. **Click "Create Web Service"**. Render will spin up a build container, install dependencies, and start the app.
7. **Visit the URL** shown in the dashboard (e.g., `https://todo-api-demo.onrender.com`).

Test a protected endpoint with a JWT you obtain from the `/login` route.

5.3 Add PM2 to the Render Build

Render runs the command you give it directly, but you can still use PM2 inside the container.

1. **Add PM2 as a production dependency**

```
npm i pm2 --save
```

2. **Create an ecosystem file** `ecosystem.config.js`:

```
// ecosystem.config.js
module.exports = {
  apps: [
    {
      name: 'todo-api',
      script: './index.js',
      instances: 'max', // one instance per CPU core (Render gives 1 core on free tier)
      exec_mode: 'cluster',
      env: {
        NODE_ENV: 'development',
      },
      env_production: {
        NODE_ENV: 'production',
      },
    },
  ],
};
```

3. **Update the start command** in `package.json`:

```
"scripts": {
  "dev": "nodemon index.js",
  "start": "pm2-runtime start ecosystem.config.js --env production",
  "postinstall": "echo 'postinstall - nothing to build'"
}
```

`pm2-runtime` is a lightweight wrapper that runs PM2 inside the container and exits when the process stops, which satisfies Render's requirement that the start command stay in the foreground.

4. **Redeploy** (push a new commit or trigger "Manual Deploy" in Render).

Render logs will now show PM2 bootstrapping the cluster and you'll see something like:

```
[PM2] Spawning PM2 daemon with pm2_home=/usr/src/app/.pm2
[PM2] App [todo-api] launched (1 instance)
```

5. **Verify** that the API still works.

5.4 Horizontal Autoscaling on Render

1. **Go to the Service Settings → Autoscaling**.
2. **Enable Autoscaling** and set:

- **Minimum Instances:** `1`
- **Maximum Instances:** `3`
- **CPU Threshold:** `70%` (Render will add a new instance when average CPU exceeds this for 2 minutes).

3. **Save**.

Render will now automatically spin up extra containers when traffic spikes. Because the app is stateless (JWT + MongoDB), any instance can serve any request.

5.5 Zero-Downtime Deploy with PM2 Reload

When you push a new version:

```
git commit -am "Add new endpoint /stats"
git push origin main
```

Render will rebuild the container, start a ***new*** PM2 process, and once it's healthy, it will replace the old instance. Under the hood, `pm2-runtime` performs a **reload** that respects existing connections.

If you want to test locally:

```
pm2 start ecosystem.config.js --env development   # first start
# make a change, then
pm2 reload ecosystem.config.js
```

You'll see **0s msdowntime** in the terminal and in a browser refresh.

5.6 Bonus: Adding a Simple Health-Check Endpoint

Many platforms ping a health endpoint to verify that the service is alive. Add this to `index.js`:

```
app.get('/health', (req, res) => {
  res.status(200).json({
    status: 'ok',
    uptime: process.uptime(),
    timestamp: Date.now(),
  });
});
```

Render automatically uses the root URL for health checks, but you can configure a custom path if needed.

5.7 Optional: Using Docker for Full Control

If you ever outgrow Render's free tier, you can ship a Docker image to **DigitalOcean App Platform**, **AWS Elastic Beanstalk**, or **Google Cloud Run**. The Dockerfile would look like:

```
# Dockerfile
FROM node:20-alpine

WORKDIR /app

COPY package*.json ./
RUN npm ci --only=production

COPY . .

ENV NODE_ENV=production
EXPOSE 8080

CMD ["pm2-runtime", "start", "ecosystem.config.js", "--env", "production"]
```

Push to a container registry, then point your cloud service at it. The same `ecosystem.config.js` and health endpoint will work unchanged.

Key Takeaways

- **Environment variables** (`dotenv`) keep secrets out of the codebase and let you switch configs between dev, staging, and prod.
- A **`Procfile`** tells most PaaS providers how to start your Node app.
- **Three beginner-friendly deployment targets**:
- **Heroku** - Git-push workflow, manual dyno scaling.
- **Render** - Autoscaling, built-in HTTPS, free tier for small APIs.
- **Railway** - Simple "Deploy from GitHub", easy variable management.
- **Process manager (PM2)**: automatically restarts crashed processes, enables cluster mode (multiple Node workers per machine), and provides zero-downtime reloads via `pm2 reload`.
- **Stateless design** (JWT + external DB) is a prerequisite for horizontal scaling.
- **Horizontal scaling** on Render/Heroku/others is achieved by adding instances (dynos, containers) behind a load balancer; **vertical scaling** means upgrading the underlying VM.
- **Health-check endpoint** (`/health`) and **global error handlers** improve reliability and give platforms a way to detect failures.
- **Autoscaling** can be configured via CPU or request thresholds; start with conservative limits on a free tier, then upgrade as traffic grows.

Quick "Deploy-and-Scale" Checklist

1. **Add `.env`** locally; load it with `require('dotenv').config()`.
2. **Create `config/config.js`** that exports all needed values.
3. **Write a `Procfile`** (`web: npm start`).
4. **Commit, push** to GitHub.
5. **Choose a platform** → connect repo → set **environment variables** in UI.
6. **Add PM2** (`pm2-runtime start ecosystem.config.js --env production`).

7. **Deploy** (Render auto-build, Heroku `git push heroku main`).
8. **Verify** `/health` and a protected route.
9. **Enable autoscaling** (Render) or **scale dynos** (Heroku).
10. **Monitor logs** (`pm2 logs`, platform dashboard).
11. **When code updates**, push → platform rebuilds → PM2 reloads with zero downtime.

Congratulations! You now have a production-ready Node.js API that can run anywhere, survives crashes, and can grow with your user base. Keep experimenting-add a Redis cache, move to Kubernetes, or integrate a CI/CD pipeline. The concepts you've mastered here are the foundation for any modern backend engineer. Happy scaling!

Summary

Summary - "Learn Node.js for Building Server-Side Web Applications" ***(≈ 750 words)***

---

1. Key Learning Outcomes

By the end of this course you should be able to:

1. **Explain the purpose and architecture of Node.js** - why a JavaScript-only runtime makes sense for I/O-heavy server-side work, and how the V8 engine, libuv, and the event loop cooperate.
2. **Set up a production-ready development environment** - install Node, use a version manager (nvm/volta), configure linting, testing, and debugging tools, and manage dependencies with npm or Yarn.
3. **Leverage core Node modules** - work comfortably with `fs`, `path`, `http`, `url`, `events`, and the Streams API while understanding how the event loop processes timers, I/O callbacks, and micro-tasks.

4. **Create and test a bare-bones HTTP server** - respond to requests, parse URLs, serve JSON, and handle basic errors without any external framework.

5. **Build robust RESTful services with Express.js** - define routes, use route parameters, query strings, and HTTP verbs; organize code with routers and controllers.

6. **Apply middleware patterns** - write reusable middleware for logging, body parsing, CORS, static-file serving, and centralised error handling.

7. **Persist data with MongoDB and Mongoose** - model schemas, perform CRUD operations, validate data, and use population/aggregation for relational-like queries.

8. **Implement authentication and security best practices** - hash passwords with bcrypt, issue JWTs, protect routes, enforce HTTPS, set secure cookies, and mitigate common attacks (XSS, CSRF, injection).

9. **Deploy a Node application to the cloud** - configure environment variables, use process managers (PM2, Docker), choose a hosting platform (Heroku, Render, AWS Elastic Beanstalk, Vercel), and understand horizontal scaling and load balancing.

---

2. Important Concepts Recap

A. Node.js & the Runtime

- **V8 Engine** executes JavaScript at native speed.
- **libuv** abstracts OS-level I/O, providing the event loop, thread-pool, and asynchronous primitives.
- **Event Loop Phases** - timers, pending callbacks, idle/prepare, poll, check, close callbacks - dictate when callbacks run. Understanding these phases helps debug latency and "callback hell".

B. Development Environment Essentials

- **Version Management** (`nvm`, `volta`) guarantees the same Node version across

machines.

- **Package Management** - `package.json` holds scripts (`npm run dev`, `npm test`), dependencies, and semantic versioning.
- **Linting & Formatting** - ESLint + Prettier enforce consistent style and catch common bugs early.
- **Testing** - Jest/Mocha + Supertest enable unit and integration tests for routes and business logic.

C. Core Modules & Asynchronous Patterns

- **File System (`fs`)** - use the promise-based API (`fs.promises`) for cleaner async/await code.
- **Streams** - readable, writable, transform, and duplex streams enable efficient handling of large payloads (file uploads, CSV parsing).
- **Events** - `EventEmitter` is the backbone of many Node APIs; custom events can decouple components (e.g., emitting "userCreated").

D. Building a Simple HTTP Server

- `http.createServer((req, res) => { ... })` gives raw access to request/response objects.
- Parsing the URL with `new URL(req.url, http://${req.headers.host})` provides pathname and query parameters.
- Proper status codes (`200`, `201`, `400`, `404`, `500`) and content-type headers (`application/json`, `text/html`) are essential for client compatibility.

E. Express.js Basics & Routing

- **Router** objects (`express.Router()`) keep routes modular (`/api/users`, `/api/posts`).
- **Route Parameters** (`/users/:id`) and **Query Strings** (`?page=2`) are accessed via `req.params` and `req.query`.
- **HTTP Verbs** (`GET`, `POST`, `PUT`, `PATCH`, `DELETE`) map naturally to CRUD operations.

F. Middleware, Error Handling & Static Files

- Middleware signature `(req, res, next)` lets you chain functionality.
- **Common middleware**: `express.json()`, `express.urlencoded()`, `cors()`, `helmet()`, `morgan`.
- Centralised error handling (`app.use((err, req, res, next) => { ... })`) ensures consistent JSON error responses and avoids duplicated try/catch blocks.
- `express.static('public')` serves CSS, images, and client-side bundles with proper caching headers.

G. MongoDB & Mongoose Integration

- **Schema Definition** - enforce field types, required constraints, default values, and custom validators.
- **Model Methods** - static (`User.findByEmail`) and instance (`user.comparePassword`) methods encapsulate business logic.
- **Population** - `populate('author')` resolves references, mimicking joins.
- **Aggregation Pipeline** - powerful for reporting (group, match, project) without pulling data into application memory.

H. Authentication & Security

- **Password Hashing** - `bcrypt.hash(password, 12)` stores a salted hash; `bcrypt.compare` verifies credentials.
- **JWT** - signed tokens (`jwt.sign(payload, secret, { expiresIn })`) provide stateless auth; protect routes with `verifyToken` middleware.
- **Secure Cookies** - `httpOnly`, `secure`, `sameSite` flags mitigate XSS/CSRF.
- **Helmet** sets HTTP headers (`Content-Security-Policy`, `X-Frame-Options`).
- **Rate Limiting** (`express-rate-limit`) and **Input Sanitisation** (`express-validator`) defend against brute-force and injection attacks.

I. Deployment, Environment Variables & Scaling

- **Environment Variables** (`process.env.PORT`, `process.env.DB_URI`) keep secrets out

of source control; use `.env` with `dotenv` locally and platform-provided config in production.

• **Process Managers** - PM2 restarts crashed processes, provides logs, and can run in cluster mode to utilise multi-core CPUs.

• **Docker** - containerises the app with a `Dockerfile`; `docker-compose` can spin up Node + Mongo together for local development.

• **Cloud Platforms** - each offers a straightforward Git-push or CI/CD pipeline; choose based on cost, region, and add-on ecosystem.

• **Scaling** - horizontal scaling (multiple instances behind a load balancer) plus a stateless design (sessions stored in Redis or JWT) ensures the app can handle increased traffic.

---

3. Next Steps Guidance

1. **Deepen Your Toolchain**

• Adopt TypeScript for static typing; it integrates seamlessly with Node and Express and catches many runtime errors at compile time.

• Explore GraphQL (Apollo Server) as an alternative to REST for flexible client queries.

2. **Build Real-World Projects**

• **E-commerce API** - product catalog, cart, checkout, payment integration (Stripe).

• **Social Media Backend** - posts, comments, real-time notifications using WebSockets (`socket.io`).

• **Microservices** - split responsibilities (auth service, user service) and communicate via a message broker (RabbitMQ, NATS).

3. **Performance & Monitoring**

• Profile with Node's built-in `--inspect` and Chrome DevTools.

• Add APM tools (New Relic, Datadog, or open-source `prom-client` + Grafana) to monitor latency, error rates, and memory usage.

4. **Security Audits**

• Run `npm audit` regularly; fix vulnerabilities with `npm audit fix` or by upgrading dependencies.

• Conduct penetration testing or use automated scanners (OWASP ZAP) to validate your defenses.

5. **Community & Continuous Learning**

• Follow the Node.js release notes; stay aware of new LTS versions and deprecations.

• Contribute to open-source libraries on GitHub; reading production code helps internalise patterns.

6. **Professional Practices**

• Write comprehensive API documentation with Swagger/OpenAPI and host it via `swagger-ui-express`.

• Implement CI/CD pipelines (GitHub Actions, GitLab CI) that run linting, tests, and automated deployments on every push.

---

4. Congratulations!

You've completed a full-cycle journey from understanding ***what*** Node.js is, through ***how*** to build and secure a modern server-side application, all the way to ***where*** to run it in production. This knowledge equips you to create performant, maintainable, and scalable back-ends that power today's web experiences.

Keep experimenting, keep building, and let the vibrant Node community be your guide. The server-side world is evolving quickly-your newly-acquired skills are the foundation for a rewarding career in full-stack development, cloud engineering, or backend architecture. Well done, and happy coding!

Glossary

Async/Await: Modern JavaScript syntax that lets you write asynchronous code that looks synchronous; it works by pausing execution until a returned **Promise** settles, then resuming with the result.

Callback: A function passed as an argument to an asynchronous API that Node invokes once the operation (e.g., I/O) completes.

Child Process: A separate operating-system process spawned from a Node program (via `child_process`), allowing CPU-intensive work to run outside the main event loop.

CommonJS: The original Node module system that uses `require()` to load files and `module.exports` to expose functionality; still the default for many legacy packages.

Event Loop: The core mechanism in Node (provided by **libuv**) that continuously processes queued callbacks, timers, and I/O events on a single JavaScript thread.

Event-Driven: A programming style where actions are triggered by events (e.g., network data arriving) rather than by sequential code execution; Node's architecture is fundamentally event-driven.

Garbage Collection: Automatic memory management performed by the **V8** engine, reclaiming objects that are no longer reachable from the program.

Global Object: The top-level namespace in a Node environment (`global`), analogous to `window` in browsers, exposing built-in APIs like `process`.

HTTP Server: A built-in Node module (`http`) that creates a server listening for incoming HTTP requests and sending responses without needing external software.

Ignition Interpreter: V8's lightweight interpreter that starts executing JavaScript quickly before the optimizing compiler (**TurboFan**) takes over.

JIT Compilation: "Just-In-Time" compilation performed by V8, converting hot JavaScript code into native machine code on the fly for better performance.

libuv: A cross-platform C library bundled with Node that provides the non-blocking I/O primitives and the event-loop implementation.

Microservice: A small, independently deployable service that performs a single business capability; Node's low overhead makes it a popular choice for building microservices.

Non-Blocking I/O: An I/O model where operations (e.g., file reads, network requests) initiate immediately and notify the event loop via callbacks or promises when finished, avoiding thread blockage.

npm (Node Package Manager): The default package registry and CLI tool for installing, publishing, and managing reusable JavaScript modules in Node projects.

Promise: An object representing the eventual completion (or failure) of an asynchronous operation, providing `then`, `catch`, and `finally` methods for handling results.

Process: The Node runtime representation of the running program (`process` global), exposing environment variables, command-line arguments, and methods to control execution.

setImmediate: A Node-specific timer that schedules a callback to run after the current poll phase of the event loop, useful for deferring work until I/O callbacks have completed.

setInterval / setTimeout: Timer functions that schedule callbacks to run repeatedly (`setInterval`) or once after a delay (`setTimeout`); they execute in the **timers** phase of the event loop.

Single-Threaded: Node executes JavaScript code on a single OS thread; concurrency is achieved via the event loop and asynchronous I/O rather than multiple threads.

TCP Socket: A low-level network endpoint provided by Node's `net` module, enabling bidirectional communication over the Transmission Control Protocol.

TLS (Transport Layer Security): Cryptographic protocol supported by Node's `tls` module for secure network communication, often used to build HTTPS servers.

V8 Engine: Google's high-performance JavaScript engine written in C++, embedded in Node to parse, compile, and execute JavaScript code.

WebSocket : A full-duplex communication protocol that runs over a single TCP connection, allowing real-time data exchange; Node libraries like `ws` and `socket.io` implement it.

Worker Threads : An optional Node feature that creates additional JavaScript threads for CPU-bound tasks, communicating via message passing to avoid blocking the main event loop.

XMLHttpRequest: A browser-only API for making HTTP requests; in Node you typically use `http`, `https`, or higher-level libraries like `node-fetch`.

Yield (via async generators): A control-flow keyword that pauses execution of an async generator, allowing the event loop to handle other work before resuming.

Zero-Delay Timer: A `setTimeout` call with a delay of `0` ms; its callback runs in the **timers** phase after the current call stack clears, but before `setImmediate`.

IMPORTANT DISCLAIMER

AI-Generated Content Notice

This document has been entirely generated by artificial intelligence technology through the Pustakam Injin platform. While significant effort has been made to ensure accuracy and coherence, readers should be aware of the following important considerations:

- The content is produced by AI language models and may contain factual inaccuracies, outdated information, or logical inconsistencies.
- Information should be independently verified before being used for critical decisions, academic citations, or professional purposes.
- The AI may generate plausible-sounding but incorrect or fabricated information (known as "hallucinations").
- Views and opinions expressed do not necessarily reflect those of the creators, developers, or any affiliated organizations.
- This content should not be considered a substitute for professional advice in medical, legal, financial, or other specialized fields.

Intellectual Property & Usage

This document is provided "as-is" for informational and educational purposes. Users are encouraged to fact-check, cross-reference, and critically evaluate all content. The Pustakam Injin serves as a knowledge exploration tool and starting point for research, not as a definitive source of truth.

Quality Assurance

While the Pustakam Injin employs advanced AI models and formatting techniques to produce professional- quality documents, no warranty is made regarding completeness, reliability, or accuracy. Users assume full responsibility for how they use, interpret, and apply this content.

Generated by: **Pustakam Injin**

Date: February 14, 2026 at 07:45 PM

For questions or concerns about this content, please refer to the Pustakam Injin documentation or contact the platform administrator.