Node.js Tutorial for Beginners: Learn Node in 1 Hour

Programming with Mosh

Below is a breakdown of the text into smaller, easy-to-understand sections with extra details to help you grasp the concepts, even if you're just starting out.

---

1. What is Node.js?

- **Definition:**
  Node.js (or simply Node) is an **open-source** and **cross-platform** runtime environment that allows you to run JavaScript code **outside of a web browser**.
- **Why It Matters:**
  Traditionally, JavaScript ran only in browsers. With Node.js, you can build server-side applications—like back-end services or APIs—that interact with databases, handle requests from web or mobile apps, and much more.
- **Real-World Use:**
  Node is especially popular for building **data-intensive, real-time applications** such as chat apps, live streaming services, and online games.

---

2. Node.js for Backend Services

- **Building APIs:**
  In many applications, the user interacts with a front-end (the part you see, like a web or mobile app), while the back-end handles things like storing data, sending emails, and processing business logic. Node.js is used to create these back-end services.
- **Scalability and Speed:**
  - **High Scalability:** Node can handle many requests at the same time. This is important for services used by lots of users simultaneously.
  - **Performance Benefits:** For example, companies like PayPal and Netflix have used Node to rebuild parts of their systems and found improvements like faster build times, less code, and the ability to serve more requests per second.
- **Advantages for Developers:**
  - **Unified Language:** If you already write JavaScript on the front-end (in your browser), you can use the same language on the back-end with Node. This makes it easier to maintain a consistent codebase.
  - **Large Ecosystem:** Node has a massive collection of open-source libraries available. This means that for many common tasks, you can use an existing module instead of writing everything from scratch.

---

3. Node.js vs. Other Technologies

- **Not a Framework:**
  Unlike frameworks such as ASP.NET, Rails, or Django, Node.js is not a framework—it is a runtime environment.

    - **Analogy:** Think of Node.js as the engine in a car. It lets JavaScript run outside the browser and interact with the computer's operating system (e.g., file system, network). Frameworks, on the other hand, are like pre-designed car models built on top of that engine.
- **Why Choose Node.js?**
    - **Ease of Use:** It's simple to get started with Node, making it great for prototyping as well as building large-scale applications.
    - **Community and Support:** Its large ecosystem means that you can often find a ready-made solution for many problems.
    - **Performance:** Node's architecture is designed for high performance in handling many concurrent connections.

---

4. Understanding the Runtime Environment

- **JavaScript Engines in Browsers:**
  Browsers like Chrome, Firefox, and Edge each have their own JavaScript engine (for example, Chrome uses **V8**). These engines convert your JavaScript into a form the computer can execute.
- **How Node.js Works:**
    - **V8 Engine:** Node.js uses Google's V8 JavaScript engine (the same one used by Chrome) to execute your code.
    - **Additional Capabilities:** Beyond running JavaScript, Node adds its own set of objects and modules (like file system and network modules) so you can perform tasks that browsers don't allow (e.g., reading files, creating servers).
- **Different Environment:**
    - **Browser Environment:** Provides objects like window and document to work with web pages.
    - **Node Environment:** Does **not** include window or document because it isn't running in a browser; instead, it provides access to the computer's file system, network, etc.

---

5. Asynchronous (Non-blocking) vs. Synchronous (Blocking) Architecture

- **Synchronous (Blocking) Architecture:**
    - **How It Works:**
      Imagine you're at a restaurant where a waiter takes your order and then stands by until your food is ready before serving anyone else.
    - **Drawback:**
      If many people are waiting for their food, the waiter can't help other tables until the current order is fulfilled. This can slow everything down.
- **Asynchronous (Non-blocking) Architecture (Node's Approach):**
    - **How It Works:**
      Think of a waiter in a busy restaurant who quickly takes your order and then moves on to help another table. When the kitchen finishes your food, the waiter is notified, and your meal is served—all without making everyone wait.
    - **Benefits:**
        - **Efficiency:** A single thread (like one waiter) can handle many requests at the same time.

        - **Scalability:** This makes Node.js ideal for applications that involve a lot of disk or network operations, allowing it to serve many clients without needing extra hardware.
- **When Not to Use Node.js:**
    - For CPU-intensive tasks (like heavy calculations, video encoding, or image processing), Node.js may not be the best choice because while it's handling one heavy task, other tasks might have to wait.

---

6. Installing Node.js and Building Your First Application

- **Installation Overview:**
    - **Checking Version:**
      You can open your command prompt (Windows) or terminal (Mac/Linux) and type:
    - node --version

      This tells you if Node is already installed and which version you have.

    - **Downloading Node.js:**
      Go to nodejs.org and download the **latest stable version**. The stable version is recommended for most users, even though there may be an "experimental" version available.
- **Creating a Simple Application:**

1. **Create a Project Folder:**

Make a new folder (for example, named first-app) for your Node project.

2. **Create a File (app.js):**

Inside your folder, create a file called app.js. This is where you will write your JavaScript code.

3. **Write Some Code:**

For instance, you can write a simple function:

```
// app.js
function sayHello(name) {
  console.log("Hello " + name);
}
sayHello("MH");
```

This code defines a function that takes a name and logs a greeting to the console.

9. **Run Your Application:**

Open your terminal, navigate to your project folder, and run:

10. node app.js

    You should see the output (e.g., Hello MH) printed on the screen.

- **A Note on Browser Objects:**
  When you try to use browser-specific objects (like window) in Node, you'll get an error because Node doesn't have these objects. Node is meant for server-side operations, so it has different built-in features (like file system access and networking).

Summary

- **Node.js** is a runtime environment that lets you run JavaScript on the server (outside the browser), making it ideal for creating backend services.
- **Backend Services and APIs:**
  Node powers the services behind web and mobile apps, handling tasks like database interactions, sending notifications, and more.
- **Advantages of Node:**
  It's easy to learn (especially if you already know JavaScript), supports high scalability through its asynchronous, non-blocking architecture, and has a vast ecosystem of libraries.
- **How It Works:**
  Node uses Google's V8 engine (like Chrome) but adds extra modules to interact with the file system and network.
- **Architecture:**
  The non-blocking model means one thread can handle many requests at once, making it efficient for data-intensive applications.
- **Getting Started:**
  You install Node from its website, check your version, create a simple app, and run it using the Node command-line tool.

This explanation covers the main points from the text while providing extra context and examples to help you understand how Node.js works and why it's popular for building scalable back-end services.

Let's break this down into smaller pieces and explain each part in simple terms. We'll cover:

1. **Global Objects in JavaScript**
2. **Differences Between Browser and Node Environments**
3. **Node's Module System**
4. **Creating and Using Your Own Modules**
5. **Using Constants and Tools for Error Prevention**

---

1. Global Objects in JavaScript

- **What Are Global Objects?**
  In JavaScript, certain objects and functions are available everywhere. For example:
  - **console**: Used to print messages to the console (using console.log()).
  - **setTimeout and clearTimeout**: Allow you to run a function after a delay (or cancel that delay).
  - **setInterval and clearInterval**: Let you run a function repeatedly at specified intervals (or stop the repetition).
- **Why "Global"?**
  Because they are available from anywhere in your code. You don't need to import or declare them—they're built into JavaScript.

---

2. Differences Between Browser and Node Environments

- **In a Browser (Client-Side JavaScript):**
  - The global object is called **window**.
  - Every global variable or function you define is a property of window.
    For example, if you write:
  - var message = "Hello";
  - console.log(window.message); // "Hello"
  - When you use console.log(), the browser actually sees it as window.console.log().
- **In Node (Server-Side JavaScript):**
  - Instead of window, Node uses a global object named **global**.
  - You can use global.console.log() or simply console.log().
  - **Important:** Variables and functions you define in a Node file (a module) are not automatically attached to the global object. They are private to that file (or module).

---

3. Node's Module System

- **What Is a Module?**
  In Node, every file is treated as a **module**. A module is like a self-contained unit of code.
  - **Private Scope:** Anything you define (variables or functions) inside that file is private to that file.
  - **Avoiding Global Clutter:** This prevents different files from accidentally overwriting each other's variables or functions.
- **The module Object:**
  Each module has a special object called module which includes:
  - **id**: A unique identifier for the module.
  - **exports**: An object where you can add variables or functions that you want to be accessible outside the module.
  - Other properties like parent, filename, children, etc. (You don't need to worry about these immediately.)
- **Why Use Modules?**
  Imagine having two files that both define a function called sayHello(). If both were global, one could overwrite the other. Modules keep code separate, so even if the same name is used, they won't conflict.

---

4. Creating and Using Your Own Modules

- **Example: Creating a Logger Module**
  Let's say you want a module dedicated to logging messages. You might create a file called logger.js:
  - **Inside logger.js:**
    - Define a variable (for example, a URL for a remote logging service).
    - Define a function called log that prints a message (or sends it to a server).
    - **Export the function:**
      You do this by assigning the function to module.exports.
    - // logger.js
    - const url = "http://mylogger.log"; // This is private to the module

        - function log(message) {
        - console.log(message); // Here, we're just printing the message for simplicity
        - }
        - module.exports.log = log; // Export the log function so other modules can use it
        - *Tip:* If you have a piece of data (like url) that is only used inside the module, you don't need to export it. This keeps your module's public interface simple.
- **Using the Module in Another File (app.js):**
    - **Import the Module:**
      In Node, you load a module using the require function. For example:
    - const logger = require('./logger'); // './logger' looks for logger.js in the same folder
    - logger.log("This is a test message"); // Calls the exported log function
    - **How It Works:**
        - The require function returns whatever is in module.exports from logger.js.
        - Since you exported an object with a log method, you can now call that method.
- **Exporting a Single Function Instead of an Object:**
  If your module only needs to provide one function, you can export it directly:
- // In logger.js
- function log(message) {
- console.log(message);
- }
- module.exports = log;

  Then, in app.js, you can call it directly:

  const log = require('./logger');
  log("This is a test message");

  This is useful for simplicity when your module has only one primary function.

---

5. Using Constants and Tools for Error Prevention

- **Using const with Modules:**
  When you import a module using require, it's best to assign it to a constant. For example:
- const logger = require('./logger');
    - **Why?**
        - This prevents accidental reassignment (changing the module reference) later in your code.
        - If you try to change a constant, you'll get an error immediately (at compile time or when using static analysis tools).
- **Static Analysis Tools (e.g., JS Hint):**
  Tools like JS Hint can check your code for common mistakes. For instance:
    - If you accidentally try to overwrite your module constant, JS Hint will alert you to the mistake before you even run your code.
    - This helps you catch errors early and write more reliable code.
- **A Quick Demo:**
  If you mistakenly write:
- const logger = require('./logger');

- logger = 1; // This will throw an error because you're trying to change a constant

  You'll see an error like "Assignment to constant variable" either during development (if using a tool) or at runtime.

Summary

- **Global objects** like console, setTimeout, and others are built into JavaScript. In browsers, they belong to the window object, while in Node, they are available on the global object.
- **Node modules** allow each file to have its own private scope. This means that variables and functions you define are not automatically accessible outside their file.
- To **share code** between files in Node, you use the module system:
  - **Export** what you want to share using module.exports.
  - **Import** it in another file using require.
- Using **constants** when importing modules can help prevent accidental overwriting, and tools like JS Hint can help catch errors early.

This structure and system make your code more modular, maintainable, and less prone to errors—especially as your application grows larger and more complex.

Let's break down this long explanation into several simple sections. We'll cover:

1. **Module Scope in Node.js**
2. **The Module Wrapper Function**
3. **Arguments Passed to the Wrapper Function**
4. **Using Built-in Modules (Path and OS)**
5. **Extra Notes and Why It Matters**

---

1. Module Scope in Node.js

- **Private by Default:**
  In Node.js, every file is considered a module. Variables and functions defined in a module are private, meaning they cannot be accessed from other files unless you explicitly export them. This avoids conflicts between different parts of your program.
- **Why This Is Useful:**
  It helps keep your code organized and prevents accidental overwriting of variables or functions from other modules.

---

2. The Module Wrapper Function

- **Behind the Scenes:**
  Node.js does not run your module code directly. Instead, it **wraps your code inside a function** before executing it. This is why when you write code in a Node file, it behaves as if it's in its own little function.

- **Illustration of the Wrapper:**
  Imagine Node.js takes your code and transforms it to something like:
- (function(exports, require, module, __filename, __dirname) {
- // Your original code goes here.
- })(exports, require, module, __filename, __dirname);
- **Key Point:**
  This wrapping mechanism is why your module's variables and functions remain private—they're contained within the function's scope.

---

3. Arguments Passed to the Wrapper Function

When Node wraps your module, it passes in several important parameters:

- **exports and module.exports:**
  - They are objects used to expose functions or variables from your module to other modules.
  - **Shortcut:** exports is just a reference to module.exports. You can add properties to exports (like exports.log = log), but you shouldn't reassign exports directly because that breaks the connection with module.exports.
- **require:**
  - This function is used to import other modules into your file.
  - Even though it seems like a global function, it's actually passed in as an argument to your module's wrapper function.
- **__filename:**
  - A string representing the complete file path of the current module.
  - Useful if you need to know exactly which file is being executed.
- **__dirname:**
  - A string representing the directory name of the current module.
  - Handy for working with relative file paths.

Extra Tip:
When you log __filename or __dirname, you see the exact file path or directory where your module lives, which can be very useful for debugging.

---

4. Using Built-in Modules (Path and OS)

Node.js comes with many built-in modules that you can use without installing anything extra. Two examples mentioned are:

The Path Module

- **What It Does:**
  The path module provides utilities to work with file and directory paths.
- **Example Usage:**
- const path = require('path'); // Load the path module.
- const parsedPath = path.parse(__filename); // Parse the current file's path.

- console.log(parsedPath);
- **Output Explanation:**
  When you log parsedPath, you get an object containing:
  - The root (e.g., "C:\" on Windows),
  - The directory,
  - The file name,
  - The file extension, and
  - The name of the file without the extension.

The OS Module

- **What It Does:**
  The os module lets you interact with the operating system. You can get information like total memory, free memory, uptime, etc.
- **Example Usage:**
- const os = require('os'); // Load the os module.
- const totalMemory = os.totalmem(); // Get total memory.
- const freeMemory = os.freemem(); // Get free memory.
- console.log(`Total Memory: ${totalMemory}`);
- console.log(`Free Memory: ${freeMemory}`);
- **Template Strings (ES6 Feature):**
  Notice the use of backticks (`) and ${}. This is called a **template string** in ES6, and it allows you to embed variables directly into strings without messy concatenation.

---

5. Extra Notes and Why It Matters

- **Module Encapsulation:**
  By wrapping code in a function, Node.js keeps each module's variables and functions safe from interference by code in other files. This is a key concept for writing large, maintainable applications.
- **Understanding the Process:**
  Knowing that Node.js wraps your module code and what parameters it passes can help you understand:
  - How and why some functions (like require) work the way they do.
  - Why you cannot reassign exports directly.
  - How Node.js gives you file path information with __filename and __dirname.
- **Real-World Implications:**
  Later in your learning journey, you'll see how to create and use your own modules, import built-in modules, and eventually build complex applications like web servers, file system utilities, or even RESTful APIs.

This explanation breaks down the inner workings of Node.js modules in a simple way. It shows you how Node keeps your code organized and private, explains the parameters provided to each module, and demonstrates how to use built-in modules to interact with the file system and operating system. With these fundamentals, you'll be better prepared to write clean, efficient Node.js applications.

Alright, let's break this down super simply so even a total beginner can understand it. 🚀

1️⃣ Understanding Modules in Node.js

🧐 What is a Module?

- A **module** is just a file that contains some code.
- You can create your own module and use it in another file.
- In Node.js, each module has its own **private scope**—this means the variables and functions inside the module are **not visible** outside of it.

2️⃣ How Does Node.js Handle Modules?

You might think that when you run a JavaScript file in Node.js, it just runs as-is, but actually, Node **wraps** your code inside a special function behind the scenes.

🤯 What is this function?

If you make a syntax error at the very first line of a module (e.g., x = ;), when you run the file, you'll see an error message along with something like this:

```
(function(exports, require, module, __filename, __dirname) {
  // Your actual module code goes here...
});
```

This is called the **module wrapper function**. Node.js does this for every module!

3️⃣ What are the Parameters in the Module Wrapper Function?

🚀 Important Stuff Inside Every Module:

1. **exports** – A shortcut for module.exports, used to export functions or variables.
2. **require** – A function to load other modules.
3. **module** – An object representing the current module.
4. **__filename** – The full path of the current file.
5. **__dirname** – The full path of the folder that contains this file.

🧐 Example: Logging __filename and __dirname

```
console.log(__filename);  // Outputs full file path
```

```
console.log(__dirname);   // Outputs full directory path
```

If you run this file with node app.js, you'll see:

/Users/luffy/Desktop/project/app.js

/Users/luffy/Desktop/project

4️⃣ Node.js Comes with Built-in Modules

Node.js has many **pre-installed modules** that help you do cool things like:

- Working with files 📂
- Running a web server 🌐
- Interacting with the operating system 💻

You don’t need to install these modules separately. Just use require() to access them.

Example:

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

console.log(path.basename(__filename)); // Output: app.js
```

5️⃣ Using the path Module

The **path** module helps us work with file paths easily.

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

const filePath = path.parse(__filename);

console.log(filePath);
```

Output:

```
{
  root: '/',
  dir: '/Users/luffy/Desktop/project',
  base: 'app.js',
  ext: '.js',
  name: 'app'
}
```

💡 **Why is this useful?**

- dir: Tells you the folder of the file.
- base: Gives you the filename with extension.
- ext: Just the file extension (.js).
- name: The filename without the extension.

6️⃣ Using the os Module

Want to know details about your computer? Use the **os** module.

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

```
console.log(`Total Memory: ${os.totalmem()}`);

console.log(`Free Memory: ${os.freemem()}`);

console.log(`OS Type: ${os.type()}`);

console.log(`Uptime: ${os.uptime()} seconds`);
```

This will show details about your system, like:

Total Memory: 17179869184

Free Memory: 8421376

OS Type: Darwin (for macOS) or Windows_NT (for Windows)

Uptime: 12345 seconds

7️⃣ Using the fs (File System) Module

The **fs (File System)** module lets you read, write, and modify files.

📝 Reading Files Synchronously (Blocking)

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

const files = fs.readdirSync('./');  // Reads files in the current directory

console.log(files);
```

This is **blocking**, meaning the program stops until the files are read.

⚡ Reading Files Asynchronously (Non-Blocking)

```
fs.readdir('./', (err, files) => {

   if (err) console.log(`Error: ${err.message}`);

   else console.log(files);

});
```

- **Why is async better?** 🚀 Because it doesn't stop the program—it keeps running while reading files.

8️⃣ Summary

- **Node wraps every module inside a function** (module wrapper function).
- **Use require() to load built-in modules** like path, os, fs, and many more.
- **Node's modules make it easy to work with files, paths, and OS info**.
- **Use asynchronous methods (fs.readFile()) instead of synchronous ones (fs.readFileSync())** to keep your app fast.

🎯 Next Steps

Now that you understand how modules work, you can start:

1. Exploring more built-in modules (http, events, etc.).
2. Learning how to create your own modules.
3. Building simple apps like a file reader, a small web server, or even a CLI tool! 🚀

Need more explanations? Just ask! 😀

Alright, let's break this down into simple terms with extra explanations so you can understand everything clearly, even if you're new to coding. 🚀

Understanding the Code Step by Step

1️⃣ What is the Logger Module?

Imagine you are making a program where you want to log (record) messages and also trigger (emit) events when something happens. To do this, we create a **Logger Module**.

A module in JavaScript (Node.js) is like a separate file that contains some reusable code. In this case, the logger module will:

- Print/log messages to the console.
- Raise (emit) an event when a message is logged.
- Allow other parts of the program to listen to that event and do something.

2️⃣ Exporting a Function in the Logger Module

The original logger module exports a simple function called log(). When you call log(), it:

1. Prints the message on the console.
2. Raises an event.

3️⃣ Using EventEmitter

Node.js has a built-in module called **EventEmitter**, which lets different parts of your program communicate with each other using events.
Think of it like a YouTube notification system:

- A YouTuber (event emitter) uploads a new video (emits an event).
- Subscribers (event listeners) get notified and watch the video (respond to the event).

So, after logging the message, we need to **emit an event** saying, "Hey, a message was logged!"

4️⃣ Fixing the EventEmitter Problem

There was a mistake in the code:

- The app.js file and the logger.js file were using **two different EventEmitter objects**.
- This is like having **two separate YouTube channels**—subscribers to one won't get notifications from the other.
- To fix this, we need to make sure both files use **the same EventEmitter object**.

5️⃣ Creating a Logger Class (Using ES6 Classes)

To solve the problem, instead of using an EventEmitter object directly, we:

1. Create a **Logger class** using ES6 syntax.
2. Use extends EventEmitter to **inherit** all features of EventEmitter.
3. Move the log() function inside the class.
4. Replace function log() with a class method (log() without the function keyword).
5. Use this.emit() inside the class to emit events.

This way, every time we create a new Logger object, it will have the ability to emit and listen for events.

The Final Logger Implementation (Fixed Version)

Before (Old Code – Two Separate EventEmitters, Problematic)

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

const emitter = new EventEmitter();

function log(message) {
    console.log(message);
    emitter.emit('messageLogged', { id: 1, url: 'http://logger.com' });
}

module.exports = log;
```

After (New Code – Using a Logger Class)

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

class Logger extends EventEmitter {
    log(message) {
        console.log(message);
        this.emit('messageLogged', { id: 1, url: 'http://logger.com' });
    }
}

module.exports = Logger;
```

Key Fixes:

✅ Logger class extends EventEmitter.
✅ The log() function is now a **method** inside the class.
✅ this.emit() is used to emit an event from the **same object**.

6️⃣ Using the Logger Class in app.js

Now, in app.js, instead of just importing a function, we import the Logger **class** and create an instance (object) of it.

```
const Logger = require('./logger');  // Importing the Logger class

const logger = new Logger();         // Creating an instance of Logger

// Register a listener for the 'messageLogged' event

logger.on('messageLogged', (arg) => {

    console.log('Listener called', arg);

});

// Call the log() method

logger.log('Hello, Node.js');
```

What Happens When We Run node app.js?

✅ It logs: Hello, Node.js
✅ Then, the event is emitted.
✅ The event listener prints: Listener called { id: 1, url: 'http://logger.com' }

Boom! 🎉 Now it works perfectly!

7️⃣ Introduction to HTTP Module

Okay, now let's switch gears to something even cooler—**creating a web server! 🌐**

What is an HTTP Server?

An **HTTP server** listens for requests from users (like when you visit a website) and responds accordingly.

In Node.js, we can use the **built-in http module** to create a basic web server.

8️⃣ Creating a Simple HTTP Server

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

const server = http.createServer((req, res) => {
    if (req.url === '/') {
        res.write('Hello, World!');
        res.end();
    }
});

server.listen(3000, () => {
    console.log('Listening on port 3000...');
});
```

Explanation:

1. We import the http module.
2. We create a server using http.createServer(), which takes a function with req (request) and res (response).
3. If the user goes to /, we send "Hello, World!" as a response.
4. We make the server listen on port **3000**.
5. We print "Listening on port 3000..." when the server starts.

2️⃣ Handling Multiple Routes

What if we want different pages for different URLs?

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

const server = http.createServer((req, res) => {
    if (req.url === '/') {
        res.write('Welcome to the Home Page!');
        res.end();
    } else if (req.url === '/api/courses') {
        res.write(JSON.stringify([1, 2, 3]));  // Send JSON data
        res.end();
```

```
    } else {
        res.write('404 Not Found');
        res.end();
    }
});
server.listen(3000, () => {
    console.log('Listening on port 3000...');
});
```

Now:

✅ Visiting http://localhost:3000/ → Shows "Welcome to the Home Page!"
✅ Visiting http://localhost:3000/api/courses → Shows [1,2,3]
✅ Visiting http://localhost:3000/anythingElse → Shows "404 Not Found"

🔟 Why Do We Use Express.js?

As you can see, handling multiple routes manually is **messy and complex**.

Instead of writing a bunch of if-else conditions, we can use **Express.js**, a popular Node.js framework.

With Express.js, the same server can be written much more cleanly:

```
const express = require('express');
const app = express();
app.get('/', (req, res) => {
    res.send('Welcome to the Home Page!');
});
app.get('/api/courses', (req, res) => {
    res.json([1, 2, 3]);
});
app.listen(3000, () => {
```

```
    console.log('Listening on port 3000...');
});
```

Why is Express.js Better?

✅ **Cleaner Code** – No more messy if-else.
✅ **Easier Routing** – Just use app.get() instead of checking req.url.
✅ **More Features** – Like middleware, request parsing, etc.

💡 **Final Recap**

1️⃣ Use **EventEmitter** for handling events in Node.js.
2️⃣ Create a **class** that extends EventEmitter to keep everything in one place.
3️⃣ Use the **HTTP module** to create a basic server.
4️⃣ Use **Express.js** for cleaner and more powerful web applications.

I hope this makes sense! Let me know if you want me to simplify any part further. 🚀😃