**Mission 3: Be a Express Mongoose Master**

**Module 12: Explore the Fundamentals of node.js**

**Lecture 12-1: How the Web Works**

When we open a website in the browser, a full process begins behind the scenes to connect our device (the client) with another computer (the server) to fetch data. This process involves several key steps: the browser sends an HTTP request to a server using the TCP/IP protocol over the internet. The request includes method type, headers, and sometimes a body. The server, which communicates using an IP address (not a domain name), processes the request and responds with an HTTP response that contains a status code, headers, and the main response body—the content we finally see on our screen. This entire interaction is possible through the client-server architecture and follows a structured request-response model.

**1. Client-Server Architecture**

The foundation of how the web works lies in the **client-server model**. A **client** is typically a web browser or application that makes a request for data. The **server** is a computer that listens for these requests and sends back responses. This structure allows centralized handling of data and logic, enabling users around the world to interact with websites and services.

**2. Request-Response Model**

The **request-response model** is the cycle through which clients and servers communicate. The client initiates a **request** to the server—this could be to fetch a web page, submit form data, etc. The server then processes that request and sends back a **response**. This loop is fundamental to every interaction on the web.

**3. Protocols**

A **protocol** is a set of rules that define how data is transmitted over the internet. One of the most important protocols in web communication is **HTTP (Hypertext Transfer Protocol)**. It governs how clients and servers format and transmit messages.

* **HTTP**: Defines how messages are formatted and transmitted between clients and servers.
* **HTTPS**: The secure version of HTTP. The “S” stands for **Secure**, and it uses **SSL/TLS encryption** to protect data from being intercepted during transfer.

**4. Domain Name, IP Address, and URL Structure**

When a user types a URL in the browser, it usually follows this format:  
protocol://domain-name/path — e.g., https://example.com/about

* **Domain Name**: A human-friendly address (like google.com) that maps to the server's actual location.
* **IP Address**: A numerical label (e.g., 172.217.160.78) that identifies a device on the internet. **Servers understand IP addresses**, not domain names.
* **DNS (Domain Name System)**: Translates domain names into IP addresses.
* **Port Number**: Specifies a particular process or service on a server. Common ones include 80 for HTTP and 443 for HTTPS.
* **Real Address**: The complete address used for communication often looks like: protocol://IP\_address:port\_number.

**5. TCP/IP and Socket Connection**

**TCP/IP** is the core communication protocol for the internet:

* **TCP (Transmission Control Protocol)**: Ensures reliable transmission of data packets by establishing a connection between client and server. It checks for errors and guarantees data arrives in the correct order.
* **IP (Internet Protocol)**: Handles the addressing and routing of data packets so they reach the correct destination.
* **Socket Connection**: A **socket** is one endpoint in a two-way communication link between two programs. TCP/IP uses sockets to establish and maintain the connection needed for web requests and responses.

**6. HTTP Request Structure**

An **HTTP request** is what the client sends to the server. It contains:

* **Request Method**: Defines the type of action. Common methods include:
  + GET: Retrieve data
  + POST: Send data (e.g., form submission)
  + PUT: Update existing data
  + DELETE: Remove data
* **Request Headers**: Key-value pairs that carry metadata, such as content type, authorization, and more.
* **Request Body**: Optional. Contains data the client wants to send to the server (typically used in POST or PUT requests).

**7. HTTP Response Structure**

Once the server processes the request, it sends back an **HTTP response**, which contains:

* **Status Code and Message**: Indicates the result of the request. Examples:
  + 200 OK: Successful
  + 404 Not Found: Resource not found
  + 500 Internal Server Error: Server failed to process request
* **Response Headers**: Provide metadata about the response, such as content type and caching information.
* **Response Body**: The main content returned by the server, such as an HTML page, JSON data, or images. This is what the user typically sees in the browser.

**Lecture 12-2: Frontend vs Backend Development**

When building a web application, two key sides work together: the **frontend** and the **backend**. The frontend is what users interact with—it runs in the browser (client-side) and is responsible for structure, style, and behavior. The backend runs on the server and handles data storage, logic, and interaction with databases. While frontend code runs in the browser after being converted to JavaScript, backend code runs on the server and can be written in multiple languages like JavaScript (via Node.js), Python, or PHP. In this model, the backend acts as a middleman between the client and the database. Websites can be either static (same content every time) or dynamic (content changes based on logic or data). Dynamic sites can be built using **Server-Side Rendering (SSR)** or **Client-Side Rendering (CSR)**, often powered by APIs that let the frontend fetch live data from the backend.

**1. Frontend and Backend Roles**

The frontend is the client-facing part of a web application. It runs in the browser and is responsible for how a website looks and responds to user actions. Frontend code (like HTML, CSS, and JavaScript) must be interpreted by the browser, so all code is ultimately converted to JavaScript before execution.

The backend, on the other hand, is the server-side logic of an application. It handles requests, processes data, interacts with databases, and sends responses. Backend code runs directly on the server and can be written in various languages such as Node.js (JavaScript runtime), Python, or PHP. Unlike frontend code, backend code is not visible or accessible to the user.

**2. Backend as the Middleman**

In a full-stack system, the backend plays the role of a middleman between the frontend and the database. When a user sends a request (like submitting a form), the frontend sends it to the backend server. The backend processes the request, interacts with the database if needed, and returns a response to the frontend.

Example:  
A file might be stored in the backend (e.g., user data), and a request is made from the frontend to access or modify it. The backend processes this request and updates or sends the data as needed.

**3. Static vs Dynamic Websites (In-Depth)**

A **static website** is made up of fixed content. Each page is pre-built using HTML and looks the same for every user. When someone visits the website, the server simply sends the existing HTML file to the browser without any processing or customization. Static websites are typically faster to load, easier to host, and secure since there's no backend logic or database connection involved. However, they cannot display user-specific data or respond to changes in real-time.

In contrast, a **dynamic website** generates content based on logic or data at the time of the request. These websites are built using both frontend and backend technologies and often connect to a database. Dynamic websites are essential when personalized content, user authentication, or interactive features are required.

There are two main ways to build dynamic websites:

**a) Server-Side Rendering (SSR)**

In **Server-Side Rendering**, the backend generates the complete HTML content based on data and logic and sends it to the browser. For example, when a user visits a news site, the server fetches the latest articles from the database, renders them into HTML using a template engine (like EJS, Handlebars, etc.), and sends that HTML to the browser. This approach is great for SEO (since search engines can read the fully rendered content) and provides a faster initial page load.

**How SSR works:**

1. Client makes a request (e.g., visiting a URL).
2. Server processes the request and fetches data if needed.
3. Server uses a **template string** to generate HTML dynamically.
4. Server sends the complete HTML response to the browser.

**b) Client-Side Rendering (CSR)**

In **Client-Side Rendering**, the server usually sends a minimal HTML file with a JavaScript bundle. The browser then fetches data from the backend using **API calls** (typically via fetch or axios), and the frontend JavaScript takes care of rendering the content. Frameworks like React, Vue, or Angular are commonly used in CSR.

**How CSR works:**

1. Client loads the initial HTML + JS from the server.
2. JS runs in the browser and makes an API request.
3. Server sends back data in JSON format.
4. JS dynamically updates the page with new content.

CSR allows rich interactivity and faster navigation after the first load, but it can affect SEO and the initial load time if not optimized properly.

**Key Differences:**

|  |  |  |
| --- | --- | --- |
| Aspect | Static Website | Dynamic Website (SSR/CSR) |
| Content | Predefined and fixed | Generated per user or logic |
| Server interaction | No backend needed | Requires backend/server |
| Performance | Very fast | Slightly slower (depends) |
| Personalization | Not possible | Easily achievable |
| Example use cases | Portfolio, Blog, Landing page | E-commerce, Dashboards, Social Media |

In modern web development, many applications use a hybrid approach—where SSR is used for initial page load and CSR is used for interactive parts of the app. This balances performance, SEO, and user experience.

**4. API and Its Benefits**

An **API (Application Programming Interface)** is a set of rules that allows the frontend to communicate with the backend. It defines how the frontend can send requests and get responses from the server or database.

**Benefits of Using APIs:**

* **Separation of concerns**: Frontend and backend can be developed independently.
* **Reusability**: APIs can be used across multiple platforms (web, mobile).
* **Scalability**: API-based systems can handle more users efficiently.
* **Flexibility**: Developers can update frontend or backend without breaking the entire system.

**5. Keyword Definitions**

* **Frontend**: The client-side of a web app that users see and interact with (HTML, CSS, JS).
* **Backend**: The server-side logic that handles data and communication with the database.
* **Client**: The browser or device that requests data.
* **Server**: The machine or system that processes client requests.
* **Node.js**: A JavaScript runtime that allows using JS on the backend.
* **Static Website**: A site with fixed content that doesn’t change based on user input.
* **Dynamic Website**: A site that updates content dynamically based on logic or data.
* **SSR (Server-Side Rendering)**: Webpage content is generated on the server and sent to the client.
* **CSR (Client-Side Rendering)**: Webpage content is generated in the browser using data from APIs.
* **API**: A bridge that allows communication between client and server.

**Lecture 12-3: Why Node js was invented**

Web development has evolved from simple, text-based pages to rich, interactive applications requiring powerful backend logic. Traditionally, full-stack developers needed to learn two different languages — one for the browser (like JavaScript) and one for the server (like PHP, Python, or Java). Node.js changed this by allowing JavaScript to run outside the browser, specifically on the server. Built on Chrome's V8 engine and written in C and C++, Node.js gave developers the ability to use a single language across both client and server. Its architecture, which includes event loops, a call stack, web APIs, and asynchronous handling via callback queues, makes it well-suited for building scalable, real-time applications. However, it's not perfect, especially for CPU-heavy tasks. Node.js remains popular due to its simplicity, performance, and large ecosystem of packages — made possible through key dependencies like V8 and libuv.

**1. Evolution of Web Development & The Birth of Full-Stack Roles**

In the early days of the internet, websites were simple documents that displayed plain **text**. Eventually, the demand for better design led to the introduction of **CSS**, which made it possible to style web pages. As user expectations grew, so did the need for **interactivity**, which gave rise to **JavaScript** — a scripting language designed to run in the browser and manipulate the **DOM (Document Object Model)**.

However, developers who wanted to build both the frontend and backend of an application had to learn two different programming languages. For instance, they used JavaScript for frontend behavior and PHP, Python, or Ruby for backend logic. This split increased the learning curve and development complexity for full-stack developers.

**2. Why Node.js Was Invented**

Node.js was created to solve this problem. It allowed **JavaScript to run on the server**, enabling developers to write both frontend and backend logic in the same language. This eliminated the need for switching between different languages and brought consistency to the development workflow. The invention of Node.js empowered JavaScript to leave the browser environment and perform tasks like reading files, handling databases, and managing servers — all of which were previously outside its reach.

**3. JavaScript Engine and Code Execution Process**

JavaScript runs inside the browser using a **JavaScript engine** — for example, Chrome uses the **V8 engine**. When JS code is executed, it goes through the following process:

**JS Code → Parser → Abstract Syntax Tree (AST) → Interpreter → Compiler → Output**

First, the parser reads the JavaScript code and converts it into an **Abstract Syntax Tree (AST)** — a tree-like representation of the code structure. The **interpreter** starts executing the code line-by-line for quick feedback, while the **compiler** (often a Just-In-Time compiler) optimizes the code for better performance during execution. The output is what the user sees or what the program returns.

**4. What Is Node.js? A JavaScript Runtime Outside the Browser**

**Node.js** is not a programming language or framework — it’s a **JavaScript runtime environment** built using **C, C++**, and the **V8 engine**. It allows JavaScript to be executed **outside the browser**, especially on the server-side. This makes it possible to use JavaScript to handle files, databases, APIs, network requests, and more — capabilities that traditional JavaScript running in the browser doesn't have.

**5. Why JavaScript Could Not Access the Server Before Node.js**

In a browser, JavaScript is sandboxed for security reasons. It only has access to the **DOM**, which allows it to manipulate HTML, CSS, and handle browser-based events. It **cannot directly access the file system, server, or databases** — features that backend languages like PHP or Python can handle. This limitation is what Node.js overcame by creating a bridge between JS and the server environment.

**6. How Node.js Executes Code: Key Components**

Node.js handles code execution using several internal components:

* **Call Stack**: This keeps track of function calls and their order.
* **Heap**: An area in memory used to store objects and variables.
* **Web APIs**: When asynchronous tasks (like timers or network requests) are initiated, they are handed over to these APIs provided by Node (not the browser).
* **Callback Queue**: When async tasks are complete, their callback functions are queued here.
* **Event Loop**: Continuously checks if the call stack is empty and if there are callbacks in the queue. If so, it moves them to the call stack to be executed.

This system allows Node.js to handle thousands of concurrent connections **without blocking**, making it perfect for real-time applications.

When Node.js runs a JavaScript program, the process starts with the **Call Stack**, where all function calls are tracked. Whenever a function is invoked, it is pushed onto the stack. If a function involves a long-running task, such as reading a file or making a network request, it doesn’t wait there. Instead, Node.js offloads this task to the **Web APIs**, which are provided by the environment (not part of JavaScript itself). These APIs handle operations like timers, HTTP requests, or file system access independently. Once the asynchronous task is completed, a **callback function** associated with that task is sent to the **Callback Queue**. Meanwhile, the **Event Loop** is constantly checking whether the call stack is empty. If it is, the event loop pushes the next function from the callback queue onto the call stack for execution. While this happens, Node.js also uses a **Heap** — a memory space — to store variables and objects. This architecture allows Node.js to handle multiple requests concurrently and efficiently without blocking the main thread, making it ideal for real-time and high-performance applications.

**7. Node.js Architecture and Dependencies: V8 and libuv**

Node.js relies on two major components:

* **V8 Engine**: Developed by Google, this is the same engine used in Chrome to execute JavaScript. It converts JS into machine code quickly and efficiently.
* **libuv**: A C library that provides **event-driven architecture** and handles **asynchronous I/O operations**. It manages the **event loop** and **thread pool**, enabling non-blocking operations such as file reads and network requests.

These components work together to give Node.js its speed and efficiency.

**8. Advantages and Popularity of Node.js**

Node.js is extremely popular because it allows **JavaScript everywhere** — on both client and server. It supports **non-blocking I/O**, has a massive package ecosystem via **npm**, and is easy to learn for developers already familiar with JS. It’s widely used in startups, real-time applications (like chat apps), REST APIs, and microservices.

**9. Limitations of Node.js**

Despite its advantages, Node.js has some limitations. It is **not suitable for CPU-intensive tasks** like image processing or complex mathematical computations. Since Node.js uses a single-threaded event loop, heavy CPU tasks can **block** the loop, degrading performance for all users. For such cases, languages like Python, C++, or Go may be more suitable.

**Lecture 12-4: High level overview of Node.js architecture**

**📌 Understanding Node.js Event-Driven Architecture**

We often hear that **Node.js is event-driven**, but what does that actually mean?

Let’s take a simple example from everyday life.

Suppose your **cousin says they’ll bring you a packet of chips**. They go to the store, buy it, and later **knock on your door**. You take the chips from them at that time.

Here, your cousin doesn’t tell you **exactly when** they'll return. Instead, **you wait for a signal (the knock)** that tells you they’ve come back. You don’t stop your life in the meantime—you go on doing other tasks.

This is exactly how **Node.js works**. It delegates a task (like reading a file or fetching data), and then **moves on to the next task** without waiting. Once the previous task is completed, **an event is triggered**, and **a callback function runs** to handle the result.

This entire flow is controlled by the **event loop**, which is often called **the heart of Node.js** because it handles almost everything internally.

**🧠 Before You Understand the Event Loop: Understand Process and Thread**

**What is a Process?**

Think of a **process** as a program that’s loaded into memory to run. For example, when you write a program, it’s just a set of instructions. It lives on your hard drive as a file, but it’s not "alive" yet.

When you run that file, your computer loads it into RAM, creates a specific environment for it to run, and this running instance is called a **process**. It includes resources like memory, registers, the program counter, and the stack.

Every process is isolated and runs independently. You can have multiple processes running at the same time, each with its own memory and context.

Now, when you run it with the command:

node server.js

Your operating system:

1. Loads the file into RAM (memory).
2. Creates a **new environment to run this program**.
3. Allocates memory, registers, and other necessary resources.

This running environment is called a **process**.

So, a **process** is:

* An independent unit of execution.
* Has its own memory, call stack, and other runtime components.
* Isolated from other processes.

🧠 You can think of a process like a hotel room that has its own setup—bed, AC, bathroom—and doesn’t share it with any other room (process).

**What is a Thread?**

A **thread** is the smallest unit of execution inside a process. A process can have one or more threads. Each thread has its own call stack, registers, and program counter.

There are two main types:

* **Single-threaded**: Only one thread runs tasks one by one.
* **Multi-threaded**: Multiple threads can run tasks in parallel.

Node.js is **single-threaded**, meaning it uses one main thread to run code. But it can still handle many tasks efficiently—thanks to its **asynchronous and non-blocking** model.

**⚙️ Node.js Runtime = Single Thread + Event Loop + Thread Pool**

When you run a JavaScript program with Node.js, you're actually running it in a special environment. This environment (Node.js runtime) is itself a **process**.

When you run a JavaScript file in Node.js:

1. A **process** is created.
2. A **single thread** is started to execute your JavaScript code.
3. Behind the scenes, **Node.js sets up an event loop** and a **thread pool (from libuv)** to handle asynchronous tasks.

Node.js runs on a **single thread**, which executes all your top-level code (outside any callback or event handler). It:

1. Loads and initializes modules.
2. Registers event listeners and callback functions.
3. Starts the **event loop**.

**🔁 Event Loop and Thread Pool**

Once everything is set up, the **event loop** starts running. It continuously checks for tasks like incoming data, timer completions, or file reads. When it finds a task ready to be executed, it picks the associated callback and runs it.

But what about heavy tasks like reading a big file or performing complex calculations?

Here’s where **the thread pool** comes in.

Node.js uses a **libuv-based thread pool** (default size: 4 threads) to offload heavy or blocking operations like file I/O. For example:

* When your code requests to read a file, the task is sent to the thread pool.
* A thread from the pool handles it without blocking the main thread.
* Once it’s done, it sends the result back to the event loop, which then executes the callback.

This way, **other users or tasks are not blocked**, and everything runs smoothly.

Let’s explain this important concept with a **real example** and **step-by-step flow**.

**🧪 Example Code:**

console.log('1');

setTimeout(() => {

console.log('2');

}, 2000);

console.log('3');

**✅ What happens when we run this?**

1. **Step 1: JavaScript engine starts execution**
   * It runs line by line.
   * Outputs 1.
2. **Step 2: setTimeout() is registered**
   * You tell Node.js: “Run this callback **after 2 seconds**.”
   * Node.js hands this off to **libuv** (a C++ library that manages the thread pool, timers, etc.).
   * The callback is registered and left alone for now.
   * Execution moves on without waiting.
3. **Step 3: Outputs 3**
   * This line is executed right after setTimeout.
4. **Step 4: Event Loop Starts Checking**
   * The main script is now done.
   * Node.js now enters the **event loop**.
   * It waits for events to happen (like timers finishing, I/O completing).
5. **Step 5: Timer expires after 2 seconds**
   * After 2 seconds, the setTimeout task is marked as **ready**.
   * Event loop picks up the callback (console.log('2')) and moves it to the **call stack**.
6. **Step 6: Callback executes**
   * 2 is printed.

**⏱ Final Output:**

1

3

2

**💡 So, What Is the Event Loop Doing?**

Think of the event loop as a **security guard** at a club:

* You (the code) tell the guard (event loop) that some VIPs (callbacks) will arrive later.
* The guard waits at the door.
* When a VIP shows up (like the result of a timer or file read), the guard lets them in one by one.
* No VIP can enter unless the guard is free to let them in (call stack is empty).

**🔄 Thread Pool – What If a Task Is Blocking?**

Suppose you have a heavy task, like reading a large file from disk.

If JavaScript tries to do it directly on the main thread, it will **block** other tasks. So instead:

* Node.js passes that task to **libuv’s thread pool** (default: 4 threads).
* One of the threads works on that task.
* When done, it **notifies the event loop**.
* The event loop picks up the callback and executes it.

**✅ This is how Node.js stays non-blocking and fast, even with just one main thread.**

**Summary**

Even though Node.js runs on a **single thread**, it:

* Uses the **event loop** to manage tasks efficiently.
* Offloads blocking operations to a **thread pool**.
* Supports **asynchronous execution**, allowing multiple tasks to progress at the same time.

This is the power of Node.js’s **event-driven architecture**—it helps you build fast, scalable applications, even with a single thread.

**Lecture 12-5: Single threaded node.js**

**Client-Server Architecture: Then and Now**

In the past, web applications often had the server handle everything. This meant the **server** contained all the **HTML, CSS, and JavaScript** needed to display the user interface. When you visited a website, the server would send you a complete, ready-to-display page. This approach worked well for browsers, but it limited flexibility. If you wanted to build a mobile app, you'd effectively need a separate server setup for it, or you'd be unable to reuse your server-side code.

Modern applications separate the **server (backend)** from the **client (frontend)**.

* The **client** can be a web browser, a mobile app (Android or iOS), or a desktop application. Its job is to handle the **user interface (UI)** and user interactions using technologies like HTML, CSS, and JavaScript.
* The **server** is purely for backend tasks. It manages **databases**, performs **heavy computations**, and handles core business logic.

When a client needs data, it sends a **request** to the server, and the server responds by sending back **JSON (JavaScript Object Notation) data**. The client then takes this JSON data and processes it to display it in its own unique UI.

The big advantage of this separation is **cross-device compatibility**. The same backend server can provide data to a browser, a mobile app, or a desktop app, as they all receive the same JSON. This makes development more efficient and allows for broader platform reach.

**Types of Server Tasks**

A server generally deals with two main types of tasks:

1. **I/O Intensive Tasks (Input/Output Intensive):** These tasks involve waiting for data to be read from or written to external sources. Examples include:
   * Fetching data from a **database**.
   * Reading or writing **files** from disk.
   * Making requests to **external APIs** (other servers).
   * Sending large amounts of data (like images) to the client. During these tasks, the server often spends more time waiting for the external operation to complete than actively computing.
2. **CPU Intensive Tasks (Central Processing Unit Intensive):** These tasks require significant processing power from the server's CPU to perform calculations. Examples include:
   * **Complex data transformations**.
   * **Image or video processing**.
   * **Encryption or decryption**.
   * Running sophisticated **algorithms**. These tasks keep the CPU busy with active computation.

**Multi-Threaded Servers: How They Work**

Traditionally, many server-side languages and frameworks operate as **multi-threaded servers**. Think of a multi-threaded server as having multiple "workers" (threads) ready to handle requests simultaneously.

When a client sends a request, a **dedicated thread** on the server picks up that request and handles its entire process. If another client sends a request at the same time, a **different thread** can handle it independently.

The main benefit here is that **threads don't block each other**. If one request is waiting for a database to respond (an I/O operation), another request, handled by a separate thread, can proceed without waiting. This allows for excellent **concurrency**.

However, multi-threaded servers usually have a **fixed number of threads**. If all threads are busy, any new incoming requests will have to **wait in a queue** until a thread becomes free. For example, if a server has 10 threads, it can handle 10 requests concurrently. An 11th request would wait. This waiting can lead to delays.

To manage this limitation, multi-threaded systems often employ **scaling strategies**:

* **Horizontal Scaling:** Adding more servers. If one server has 10 threads, adding another server doubles the capacity to 20 threads. This means more machines, more setup, and higher costs.
* **Vertical Scaling:** Upgrading a single server's resources, like adding more CPU cores or RAM. For instance, upgrading an 8-core server to a 32-core server. This increases the power of a single machine, but higher-end hardware is very expensive.

Both scaling methods lead to significantly increased costs and resource consumption.

**Node.js: Single-Threaded with Non-Blocking I/O**

Node.js is unique because it's fundamentally a **single-threaded runtime**. This means your main JavaScript code runs on just one thread. At first, this seems problematic: how can one thread handle many requests concurrently without blocking?

Node.js solves this with a brilliant design: **non-blocking I/O**, powered by an **Event Loop** and an underlying **Thread Pool**.

1. **I/O Intensive Tasks:** When Node.js encounters an I/O operation (like fetching data from a database or reading a file), it doesn't execute that operation on its single main thread. Instead, it **offloads** the task to an internal **Thread Pool** (managed by libuv under the hood). The main Node.js thread immediately becomes **free** to process the next incoming request. It doesn't wait for the I/O to complete.
2. **Asynchronous Processing:** Once the task in the Thread Pool finishes (e.g., the database returns data), it notifies the **Event Loop**. The Event Loop then places a "callback" for that completed task into a queue. When the main Node.js thread finishes its current execution, the Event Loop picks up the next callback from the queue and executes it. This is **asynchronous** execution.

This mechanism is called **Non-Blocking I/O** because even though there's only one main thread, I/O operations don't block subsequent requests. Many I/O requests can be initiated and handled concurrently by the Thread Pool, while the main thread remains responsive.

However, there's a "catch" with Node.js and **CPU Intensive Tasks**: Historically, CPU-intensive tasks **run directly on the main Node.js single thread**. If a complex calculation takes a long time (e.g., 10 seconds), the **entire Node.js application will be blocked** during that period. No other requests (even I/O ones) can be processed until that heavy computation finishes. This is why Node.js was traditionally not recommended for applications that are primarily CPU-bound.

**Modern Solution: worker\_threads** Fortunately, recent Node.js versions introduced the **worker\_threads module**. This allows developers to explicitly create and manage additional threads within a Node.js application. Now, you can offload those CPU-intensive tasks to these dedicated worker threads, ensuring the main Node.js thread remains free and non-blocked.

This means Node.js can now effectively handle both I/O intensive tasks (via the Event Loop and Thread Pool) and CPU intensive tasks (via worker\_threads), making it a highly versatile and powerful runtime. The **Event Loop** continues to be the core orchestrator, ensuring smooth, non-blocking operations.

**Overall Summary:** Traditional servers often bundle client and server logic, typically using a multi-threaded approach to handle concurrent requests. While multi-threading prevents blocking between requests, it faces limitations in scalability without significant cost increases. Node.js, a single-threaded runtime, overcomes this by using non-blocking I/O: it offloads I/O tasks to a separate thread pool, keeping its main thread free for new requests, orchestrated by an Event Loop. For CPU-intensive tasks, which previously blocked the single thread, Node.js now offers worker\_threads to enable true multi-threading, expanding its capabilities for a wider range of applications.

**Lecture 12-6: How Event Loop Works**

**The Event Loop: Orchestrating Node.js's Concurrency**

As we know, Node.js is a single-threaded runtime. This means your main JavaScript code executes on a single thread. However, Node.js can still handle many operations concurrently, especially I/O intensive tasks. This is primarily managed by the **Event Loop** and an underlying **Thread Pool**.

When an I/O intensive task (like a database query or file read) is encountered by the single thread, it's offloaded to the **Thread Pool** for execution. Once the task in the Thread Pool completes, the result is sent back to the single thread, which then prepares the response for the user. The entire process of offloading tasks to the Thread Pool, completing them, and sending back responses is orchestrated by the **Event Loop**.

The Event Loop operates on an **event-driven architecture**. This means it constantly "listens" for events (like a new request coming in or an I/O operation completing). When an event is "triggered," the Event Loop takes action:

1. It sends the task to the Thread Pool if it's an I/O operation.
2. Once the Thread Pool finishes the task, it signals back to the Event Loop.
3. The Event Loop then pushes the result (via a callback function) to be processed by the single thread.
4. Finally, the single thread sends the response back to the user.

The Event Loop is a continuous process that runs as long as the Node.js application is active. It acts like a "watchman," always observing for triggered events and ensuring they are handled efficiently without blocking the main single thread.

**How the Event Loop Works: Phases and Queues**

The Event Loop executes in distinct "phases," constantly looping through them. When a Node.js process starts, the Event Loop activates and begins this continuous cycle. It also manages several "queues" where operations wait to be processed. The most important one is the **Callback Queue**.

The Event Loop's operational flow involves several phases, executed in a specific order within each cycle:

1. **Timers Phase (setTimer callbacks):**
   * The Event Loop first checks for any **setTimeout()** or **setInterval()** callbacks that are due to execute. These are functions scheduled to run after a certain delay.
   * These timers have the highest priority in this specific cycle of the event loop.
2. **I/O Callbacks Phase (I/O polling and callbacks):**
   * After processing timers, the Event Loop moves to this phase.
   * It checks for and executes callbacks related to **I/O operations** that have completed (e.g., database query results, file read completions, network requests). These are the tasks that were initially offloaded to the Thread Pool.
3. **setImmediate() Callbacks Phase:**
   * This phase handles callbacks scheduled by **setImmediate()**. These are functions that are executed immediately after the current phase of the Event Loop completes.
4. **Close Callbacks Phase (close events):**
   * Finally, this phase handles any close event callbacks, such as those triggered when a socket or handle is unexpectedly closed.

**Between these main phases**, the Event Loop also processes special queues:

* **process.nextTick() Queue:** Callbacks in this queue have the **highest priority** and are executed immediately after the current operation on the call stack completes, *before* the Event Loop moves to the next phase. If you have a process.nextTick() call, it will run even before setImmediate() or setTimeout().
* **Microtask Queue (Promises, queueMicrotask):** This queue holds callbacks for resolved Promises and tasks scheduled with queueMicrotask(). These microtasks are also executed right after the current operation on the call stack finishes, and before the Event Loop proceeds to the next main phase. They generally run *after* process.nextTick() callbacks.

Understanding these phases and queues is crucial for comprehending how Node.js manages concurrency and the order in which different types of operations are processed. While you might not always need to manipulate these directly in daily coding, knowing this architecture is vital for advanced Node.js development, debugging performance issues, and customizing its behavior.

**Overall Summary:** The Node.js Event Loop orchestrates its single-threaded, non-blocking I/O model. It continuously cycles through distinct phases (Timers, I/O Callbacks, setImmediate, Close Callbacks) to process events. I/O tasks are offloaded to an internal Thread Pool, and their results return to the Event Loop, which then queues their callbacks for the single thread. High-priority process.nextTick() and Microtask (Promises) queues are processed between main Event Loop phases, ensuring efficient and non-blocking execution flow. Understanding these phases and queues is key for advanced Node.js development.

**Lecture 12-7: Install Node js using fnm**

Let's break down the process of installing Node.js using FNM (Fast Node Manager) and understand Node.js versioning.

**Understanding Software Versioning**

In the digital world, we constantly work with various software and packages, from mobile apps to developer tools. You'll notice that every piece of software you use has a **version number**.

Think about your mobile phone: your Android operating system frequently gets new updates, right? Or if you're a Windows user, you get Windows updates. When you install a Facebook update from the Play Store, it often comes with security fixes or new features. Each of these updates signifies a change, and these changes are tracked by a **version number**. This numbering helps us identify different iterations of a software.

**Node.js Versioning Explained**

Node.js, being a piece of software, also follows a versioning scheme. It's common to work with different Node.js versions, so understanding its versioning is crucial. A typical Node.js version, for example, 20.11.1, is divided into three parts separated by dots:

1. **Major Version (e.g., 20):**
   * This number indicates a **major change** in the software.
   * Major changes often introduce **breaking changes**, meaning features that worked in previous major versions might not work the same way (or at all) in the new major version.
   * For example, when a significant new system or paradigm was introduced into Node.js, it might trigger a major version increment.
   * Node.js version 20 is a Long Term Support (LTS) version, meaning it receives extended maintenance. Node.js version 24 is a newer major version.
2. **Minor Version (e.g., 11):**
   * This number signifies **minor changes** or **improvements** to existing features.
   * Minor versions typically add new functionalities in a backward-compatible way, meaning they shouldn't break existing code.
   * For instance, an optimization or enhancement to an existing feature within the Node.js 20 series would increment the minor version (e.g., from 20.10.x to 20.11.x).
3. **Patch Version (e.g., 1):**
   * This number represents **small updates**, primarily for **bug fixes** or **security patches**.
   * Patch versions are meant to be fully backward-compatible and address critical issues without introducing new features or breaking existing ones.
   * For example, a quick security update to Node.js 20.11.0 would result in version 20.11.1.

These three numbers combined give a complete picture of a software's version, indicating the scope of changes it contains.

**Installing Node.js Using FNM on Windows**

The lecture focuses on installing Node.js using **FNM (Fast Node Manager)**, a tool that makes managing multiple Node.js versions easy. The example given is for Windows.

Here's a breakdown of the installation steps and common issues:

1. **Install FNM via PowerShell:**
   * The first step is to open PowerShell and run the command: wget install-fnm.now.sh -OutFile install-fnm.ps1; & install-fnm.ps1 (Note: The lecture used wget install.fnm.now.sh -OutFile install-fnm.ps1; & install-fnm.ps1 but the official FNM script is https://fnm.vercel.app/install).
   * This command downloads the FNM installation script and then executes it, installing FNM on your system.
2. **Verify FNM Installation:**
   * After installation, you might try to check the FNM version by typing fnm --version in PowerShell.
   * **Common Issue:** You might receive an error stating that fnm is not recognized.
   * **Solution:** The installer often suggests closing and reopening your PowerShell terminal. This is because environment variables (which tell your system where to find fnm) need to be reloaded. After restarting PowerShell, fnm --version should display the version.
3. **Locating FNM Installation Path (for manual environment variable setup):**
   * FNM typically automatically adds its path to your system's environment variables. However, if it doesn't, you might need to add it manually.
   * To find the path, navigate to: C:\Users\<YourUsername>\AppData\Local\fnm. (Note: AppData is usually a hidden folder, so you might need to enable "Show hidden items" in File Explorer's View tab.)
   * Inside the fnm folder, you'll likely find the fnm.exe executable. The path to this folder (e.g., C:\Users\<YourUsername>\AppData\Local\fnm) should be added to your system's Path environment variable. This ensures your command line can find the fnm command.
4. **Installing a Node.js Version with FNM:**
   * Once FNM is installed and accessible, you can install a specific Node.js version. For example, to install Node.js version 20, you would use: fnm install 20.
   * FNM will download and install the specified Node.js version.
5. **Setting a Default Node.js Version:**
   * After installing a Node.js version, you can set it as the default for your system: fnm default 20.
   * Now, when you type node --version in your terminal, it should show the default version you set (e.g., v20.x.x).
6. **Managing Multiple Node.js Versions:**
   * FNM's real power comes from easily switching between different Node.js versions.
   * To see all installed Node.js versions, use: fnm list.
   * To switch to a different version, use: fnm use <version\_number> (e.g., fnm use 24). This temporarily changes the Node.js version for your current terminal session.
   * To set a new default version, use fnm default <version\_number>.

Using FNM simplifies managing multiple Node.js versions on your machine, which is incredibly useful for developers working on different projects that might require specific Node.js environments.

**Overall Summary:** Software versioning uses major, minor, and patch numbers to indicate the scope of changes (breaking, new features, bug fixes, respectively). Node.js follows this. Installing Node.js with FNM (Fast Node Manager) on Windows streamlines version management. After installing FNM via PowerShell, you can use commands like fnm install <version>, fnm default <version>, and fnm use <version> to easily install, set default, and switch between different Node.js versions, which is essential for developers working with varied project requirements.

**Lecture 12-8: Modular System in node js**

**The Evolution of JavaScript Scope and Organization**

In the early days of web development, we would create a website's structure with HTML, beautify it with CSS, and add interactivity using JavaScript. Back then, when declaring variables in JavaScript, we commonly used the var keyword.

**The Problem with var and Global Scope**

The main issue with var was that variables declared with it were often **globally accessible**. Global scope refers to a top-level object in JavaScript (like the window object in browsers) where many built-in functions (like setTimeout, setInterval) and globally declared variables reside.

For instance, if you declared var myVariable = 10; or a function var myFunction = function() { ... };, both myVariable and myFunction would be accessible directly through window.myVariable and window.myFunction (in a browser environment). This means any script, anywhere on the page, could potentially access or modify these variables and functions.

This global accessibility led to significant problems:

* **Naming Conflicts:** If multiple JavaScript files were included in an HTML page, and each file declared a variable or function with the same name using var, they would **overwrite** each other, leading to unexpected behavior and bugs.
* **Lack of Encapsulation:** It was difficult to create self-contained pieces of code, as everything could potentially be accessed or modified by other parts of the application.

**Introducing let and const for Block Scope**

To solve these problems, ES6 (ECMAScript 2015) introduced let and const. Variables declared with let or const are **block-scoped**, meaning they are only accessible within the block of code (e.g., inside curly braces {}) where they are defined, not globally.

For example, if you declare let myVariable = 10; , myVariable will *not* be accessible via window.myVariable in the global scope; attempting to do so would result in undefined. This fundamental change helped prevent naming conflicts and provided better encapsulation for variables.

**Immediate Invoked Function Expressions (IIFEs)**

Even before let and const became widely adopted, developers used a pattern called **Immediately Invoked Function Expressions (IIFEs)** to create isolated scopes. An IIFE is a function that is defined and executed immediately.

**Example of an IIFE:**

(function() {

let myLocalVariable = 10; // This variable is only accessible inside this IIFE

function add(a, b) {

return a + b;

}

console.log(add(myLocalVariable, 5));

})();

If you had two separate JavaScript files, each with an IIFE, and both IIFEs declared a variable named myLocalVariable or a function named add, they would not conflict because each exists within its own isolated function scope. This was a good way to prevent global namespace pollution.

While IIFEs were effective for preventing conflicts, they become challenging to manage and organize as an application grows very large. It's not practical to wrap every single piece of code in an IIFE. This led to the development of **modular systems**.

**Modular Systems in Node.js**

Modular systems allow you to break down your code into small, independent, and reusable units called **modules**. Each module is a self-contained "chunk of code" that encapsulates its own variables and functions, preventing global conflicts and promoting better organization.

Node.js primarily supports two main modular system concepts:

1. **CommonJS (CJS):** This is the original and default module system in Node.js.
   * **Exporting:** To make code available from one file to another, you use the module.exports object. You assign what you want to export (variables, functions, objects) to module.exports or exports.

// myModule.js

const PI = 3.14;

function add(a, b) {

return a + b;

}

module.exports = { PI, add }; // Exporting multiple items

* + **Importing (Requiring):** To use code from another module, you use the require() function.

// main.js

const { PI, add } = require('./myModule.js'); // Importing from myModule.js

console.log(add(PI, 5));

* + **File Naming:** CommonJS modules typically use the .js file extension.
  + **Analogy:** Think of CommonJS like a system where one country (your module file) explicitly **exports** a product (your code) to another country (another module file) that then **requires** or imports that product to use it.

1. **ECMAScript Modules (ESM or ES Modules):** This is the standardized module system for JavaScript, now also fully supported in Node.js.
   * **Exporting:** You use the export keyword.

// myModule.mjs (or .js if configured)

export const PI = 3.14;

export function add(a, b) {

return a + b;

}

// export default someDefaultValue; // For default exports

* + **Importing:** You use the import keyword.

// main.mjs (or .js if configured)

import { PI, add } from './myModule.mjs'; // Importing specific items

// import myDefaultValue from './myModule.mjs'; // For default imports

console.log(add(PI, 5));

* + **File Naming:** ES Modules typically use the .mjs file extension, or you can configure package.json to treat .js files as ESM.

Both CommonJS and ES Modules serve the same purpose: to make code in one file accessible and reusable in another file, while maintaining isolation and preventing global scope conflicts.

**Types of Modules in Node.js**

Node.js applications commonly use three types of modules:

1. **Local Modules:**
   * These are modules you **create yourself** within your project.
   * You write your own JavaScript files, define functions or variables, and then export them (using module.exports for CommonJS or export for ESM).
   * Then, in other files within your project, you require() or import these exported items to use them.
   * **Example:** If you have math.js with an add function, you'd export add from math.js and then require or import it into your app.js file.
2. **Built-in Modules:**
   * These are modules that come **pre-installed with Node.js** itself. They provide core functionalities like file system operations (fs), network communication (http), path manipulation (path), etc.
   * You don't need to install them; you just require() or import them directly by their name.
   * **Example:** const fs = require('fs'); or import http from 'http';
3. **Third-party Modules:**
   * These are modules created by **other developers** and are typically published to a package registry like **npm (Node Package Manager)**.
   * You install them into your project using npm install <package-name>.
   * Once installed, you can require() or import them into your project just like built-in modules.
   * **Example:** const express = require('express'); (after running npm install express)
   * These third-party packages themselves are often collections of modules, and the developers of these packages have already handled exporting their functionalities for you to use. You simply install and import.

Modular systems are fundamental to building scalable, maintainable, and organized Node.js applications. They allow developers to create clean, reusable code, avoiding the "spaghetti code" that can result from global scope pollution.

**Lecture 12-9: Explore commonjs Module**

Let's explore **CommonJS** and **Local Modules** in Node.js, focusing on how you create and use your own code modules.

**Understanding Local Modules in Node.js**

In Node.js, we primarily encounter three types of modules:

1. **Local Modules:** Modules we write ourselves for our specific project.
2. **Built-in Modules:** Modules provided by Node.js itself (e.g., fs, http).
3. **Third-party Modules:** Modules created by other developers and installed via npm (e.g., express).

A **local module** is essentially a piece of code written in one file that you want to make available and reusable in other files within your project. When you "export" code from a file, you make it accessible to the "outside world" (other files). That file then acts as a module.

Let's walk through a practical example to understand this.

**Scenario: Code in Separate Files**

Imagine you have two files: file1.js and file2.js.

**file2.js:**

// file2.js

const a = 'Hello';

// We want to use 'a' in file1.js

**file1.js :**

// file1.js

console.log(a); // Trying to access 'a' directly

If you try to run file1.js using node file1.js, you'll get an error like ReferenceError: a is not defined. This happens because file1.js has no knowledge of variables defined in file2.js. Each file initially exists in its own isolated scope.

**Exporting and Requiring Local Modules with CommonJS**

To make a from file2.js accessible in file1.js, we need to use the CommonJS module system's module.exports for exporting and require() for importing.

**1. Exporting from file2.js**

To make the variable a available, you need to export it from file2.js. The module.exports object is what a module exports and what require() will return.

// file2.js

const a = 'Hello';

module.exports = a; // Exporting the variable 'a'

Here, we're assigning the value of a directly to module.exports. This means when another file requires file2.js, it will receive the string 'Hello'.

**2. Requiring in file1.js**

Now, in file1.js, you can use require() to bring in the exported value from file2.js.

const paramOne = require('./file2.js'); // The path is relative to file1.js

console.log(paramOne); // This will now log 'Hello'

When you run node file1.js, you will now see Hello logged to the console. This demonstrates that file1.js successfully "required" and accessed the exported value from file2.js.

**Understanding module.exports and Exporting Multiple Items**

It's important to understand what module.exports truly is. In Node.js, every JavaScript file is treated as a module. Each module has a module object, and this object has an exports property. By default, module.exports is an empty object ({}).

If you assign a single value directly to module.exports, that value becomes the *entire export* of the module. For example, if module.exports = 'Hello';, then require('./file2.js') will simply return the string 'Hello'.

**Exporting Multiple Items:** Often, you'll want to export multiple variables, functions, or objects from a single module. You do this by adding them as properties to the module.exports object.

Let's modify file2.js to export both a variable a and a function add:

// file2.js

const a = 10; // Changed 'Hello' to 10 for addition example

const b = 20; // Another variable

function add(param1, param2) {

return param1 + param2;

}

// Exporting multiple items as properties of an object

module.exports = {

myVariableA: a,

myVariableB: b,

addFunction: add

};

Now, when file1.js requires file2.js, it will receive an object containing myVariableA, myVariableB, and addFunction.

// file1.js

const myModule = require('./file2.js');

console.log(myModule.myVariableA); // Accessing the exported variable 'a'

console.log(myModule.addFunction(myModule.myVariableA, 5)); // Calling the exported function

console.log(myModule.myVariableB); // Accessing another exported variable

When you run node file1.js, you'll see:

10

15

20

This shows that you can successfully export and access multiple pieces of code from one module.

**Destructuring Exports for Cleaner Code**

Accessing exported items using dot notation (e.g., myModule.myVariableA) is perfectly fine. However, for cleaner code, especially when you need to access several exported items, you can use **object destructuring** when you require() the module.

// file1.js

// Using object destructuring to directly get the exported properties

const { myVariableA, addFunction, myVariableB } = require('./file2.js');

console.log(myVariableA); // Directly access myVariableA

console.log(addFunction(myVariableA, 5)); // Directly call addFunction

console.log(myVariableB); // Directly access myVariableB

This achieves the same result but makes the code in file1.js more concise and readable, as you directly access the exported names without needing the myModule. prefix.

This covers the fundamentals of CommonJS and creating local modules. By using module.exports to export code and require() to import it, you can effectively organize your Node.js applications into reusable and manageable modules, preventing global scope conflicts and making your code more maintainable.

**Lecture 12-10: Name Exports, Name Aliasing & Index Export**

**Review: CommonJS Exports**

In the previous video, we learned how to export variables and functions from one file (file2.js) and then require() them in another (file1.js). We typically used module.exports = { ... } to export multiple items as properties of an object.

**file2.js (Example from previous lecture)**

// file2.js

const a = 10;

const b = 20;

function add(param1, param2) {

return param1 + param2;

}

module.exports = {

myVariableA: a,

myVariableB: b,

addFunction: add

};

**file1.js (Example from previous lecture)**

// file1.js

const { myVariableA, myVariableB, addFunction } = require('./file2.js');

console.log(myVariableA); // Output: 10

console.log(myVariableB); // Output: 20

console.log(addFunction(2, 3)); // Output: 5

**Name Aliasing with CommonJS require()**

Sometimes, when you require a module, the names of the exported variables or functions might conflict with existing names in your current file, or you might simply prefer a different name for clarity. This is where **name aliasing** comes in handy.

With object destructuring, you can easily rename imported variables.

**Scenario:** Suppose you have file3.js that also exports a variable named a, but you want to import it into file1.js without conflicting with myVariableA from file2.js.

**file3.js**

// file3.js

const a = 40; // Different 'a'

function subtract(x, y, z) { // A new function with same name as 'add' in file2, but takes 3 params

return x - y - z;

}

module.exports = {

myVariableA: a,

subtractFunction: subtract

};

**file1.js (with Aliasing)**

// file1.js

// Importing from file2.js

const { myVariableA, myVariableB, addFunction } = require('./file2.js');

// Importing from file3.js with aliasing

// Renaming 'myVariableA' from file3.js to 'aFromFile3'

// Renaming 'subtractFunction' from file3.js to 'subtractFromFile3'

const { myVariableA: aFromFile3, subtractFunction: subtractFromFile3 } = require('./file3.js');

console.log(`From file2.js: A = ${myVariableA}, B = ${myVariableB}, Add(2,3) = ${addFunction(2, 3)}`);

console.log(`From file3.js: A = ${aFromFile3}, Subtract(10,2,3) = ${subtractFromFile3(10, 2, 3)}`);

When you run node file1.js, the output would be:

From file2.js: A = 10, B = 20, Add(2,3) = 5

From file3.js: A = 40, Subtract(10,2,3) = 5

**Why is aliasing useful?** You might wonder why we need aliasing if we can just rename variables in our own files. The primary reason is when you're working with **third-party packages or built-in Node.js modules**. You cannot change the variable names within those external modules. If an external module exports a variable with a name that conflicts with one you already have or want to use differently, aliasing provides a clean way to handle the conflict without modifying the original module or your entire codebase.

**Index Export (Entry Point for Folders)**

As your project grows, you might organize related modules within a folder. For example, you might have an utils folder containing various utility functions like add.js and subtract.js.

**Folder Structure:**

my-project/

├── utils/

│ ├── add.js

│ └── subtract.js

└── main.js

**utils/add.js**

JavaScript

// utils/add.js

function add(param1, param2) {

return param1 + param2;

}

module.exports = { add }; // Exporting as an object with 'add' property

**utils/subtract.js**

JavaScript

// utils/subtract.js

function subtract(param1, param2) {

return param1 - param2;

}

module.exports = { subtract }; // Exporting as an object with 'subtract' property

Now, in main.js, you might want to import both add and subtract. You could do this by requiring each file individually:

**main.js (Without Index Export)**

// main.js

const { add } = require('./utils/add.js');

const { subtract } = require('./utils/subtract.js');

console.log(add(3, 2)); // Output: 5

console.log(subtract(3, 2)); // Output: 1

This works, but as you add more utility files, your main.js can become cluttered with many require statements. This is where **index.js (or index.mjs for ESM)** comes in as a convention for **index export**.

Node.js has a special behavior: if you require() a directory, it will automatically look for an index.js (or index.json, index.node) file inside that directory and load it. We can leverage this to create a single entry point for our utils folder.

**Create utils/index.js:**

// utils/index.js

// Re-exporting modules from other files in the 'utils' folder

const { add } = require('./add.js');

const { subtract } = require('./subtract.js');

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

Now, your folder structure looks like this:

my-project/

├── utils/

│ ├── add.js

│ ├── subtract.js

│ └── index.js <-- New index file

└── main.js

Now, in main.js, you can require the entire utils folder, and Node.js will automatically load utils/index.js.

**main.js (With Index Export)**

// main.js

const { add, subtract } = require('./utils'); // Note: No '/index.js' needed!

console.log(add(3, 2)); // Output: 5

console.log(subtract(3, 2)); // Output: 1

**Benefits of Index Export:**

* **Cleaner Imports:** Your importing files become cleaner as you only need one require statement for a group of related functionalities.
* **Better Organization:** It provides a clear entry point for a logical collection of modules within a folder.
* **Easier Maintenance:** If you add or remove utility functions, you only need to update utils/index.js, not every file that uses those utilities.

This approach makes your code more organized, manageable, and easier to maintain in larger projects.

**Overall Summary:** We further explored Node.js's CommonJS module system by discussing **name aliasing** during require() destructuring, which allows renaming imported variables/functions to avoid conflicts, especially with external modules. We then learned about **index export** using an index.js file within a directory. This pattern serves as a single entry point for a collection of related modules, simplifying require() statements and enhancing code organization and maintainability.

**Lecture 12-11: IIFE A Module Wrapper**

Let's explore an interesting concept in Node.js: how it wraps your code in an **IIFE (Immediately Invoked Function Expression)**, effectively creating a module scope for each file.

**Global Objects: Browser vs. Node.js**

Before diving into Node.js's module wrapper, let's briefly revisit **global objects** in JavaScript environments.

**Browser Environment: window Object**

When you run JavaScript in a web browser, there's a global object called window. This window object holds all global variables, functions, and properties available to your JavaScript code. For example, when you use document.getElementById(), fetch(), setTimeout(), or setInterval(), you're implicitly accessing methods on the window object. You don't have to explicitly type window.setTimeout() because window is the default global context.

// In a browser's console

console.log(window.document); // Accesses the Document Object Model

console.log(window.fetch); // The global fetch function

console.log(window.setTimeout); // The global setTimeout function

All your globally declared variables and functions in a browser are attached to this window object.

**Node.js Environment: global Object**

Node.js, being a server-side runtime, doesn't have a window object. Instead, it has its own global object called global. Similar to the browser's window, the global object in Node.js provides access to globally available functions and properties like setTimeout, setInterval, clearTimeout, and clearInterval.

// In a Node.js file (e.g., globalTest.js)

console.log(global.setTimeout); // The global setTimeout function in Node.js

console.log(global.console); // The global console object

However, a key difference arises: require, module, and exports (which are crucial for Node.js's modular system) are **not** properties of the global object. If you try to log global.require or global.module, you'll find them to be undefined.

So, how does Node.js provide access to require, module, and exports within each file? This is where the IIFE module wrapper comes into play.

**The IIFE Module Wrapper in Node.js**

Node.js elegantly solves the problem of providing require, module, exports, \_\_filename, and \_\_dirname to each file without polluting the global scope. It does this by wrapping the entire content of each JavaScript file in an **Immediately Invoked Function Expression (IIFE)**.

**What is an IIFE?**

An IIFE is a function that is defined and executed immediately after its creation. The general syntax looks like this:

(function() {

// Your code here

})();

The outer parentheses () around the function declaration make it a function expression, and the final () immediately invokes it.

**How Node.js Uses IIFEs as Module Wrappers:**

When Node.js loads a JavaScript file, it doesn't execute the file's code directly. Instead, it effectively transforms your file's content into a function and then immediately calls that function. This "wrapper function" looks something like this:

(function(exports, require, module, \_\_filename, \_\_dirname) {

// Your actual file's code goes here

// For example:

// const myVar = 10;

// module.exports = myVar;

})(exports, require, module, \_\_filename, \_\_dirname);

Let's break down why this is brilliant:

1. **Scope Isolation:** Each file's code runs within the scope of this wrapper function. This means variables and functions declared with var, let, or const inside your module file are **not globally accessible** (they are scoped to this IIFE), preventing naming conflicts between different files. This is similar to how IIFEs were used in browsers to create private scopes.

// myModule.js

const myLocalVariable = 10; // This is local to myModule.js

// console.log(global.myLocalVariable); // Would be undefined if executed in another file

module.exports = myLocalVariable;

1. **Providing Module-Specific Variables:**
   * The exports, require, module, \_\_filename (current file's path), and \_\_dirname (current directory's path) are passed as **parameters** to this wrapper function.
   * Because they are parameters, they are accessible within the scope of your file's code. This is why you can use require(), module.exports, etc., directly in your Node.js files without them being part of the global object. They are effectively local variables for each module file.

**Example:**

// exampleModule.js

console.log(require); // This is the 'require' function passed as a parameter

console.log(module); // This is the 'module' object passed as a parameter

console.log(\_\_filename); // Full path to this file

console.log(\_\_dirname); // Full path to this file's directory

const greeting = "Hello from exampleModule!";

module.exports = greeting;

If you run this file with node exampleModule.js, you'll see the respective functions, objects, and paths printed. They are not coming from global.

1. **Encapsulation:** This wrapper effectively turns every Node.js file into a self-contained module. Everything inside the IIFE is private to that module unless explicitly exported via module.exports.

In essence, Node.js's use of IIFEs as module wrappers provides a robust and clean modular pattern. Unlike older browser JavaScript (without module systems), where all script variables could pollute the global window object, Node.js ensures that each file operates in its own isolated scope, receiving necessary module-specific variables as function parameters. This is a foundational aspect of Node.js's powerful and organized module system.

**Lecture 12-12: ES Module & Module Summary**

Let's dive deeper into Node.js's module system, specifically focusing on **ECMAScript Modules (ESM)**, and then summarize the key concepts we've covered.

**ECMAScript Modules (ESM) in Node.js**

We've already explored **CommonJS (CJS)**, the traditional module system in Node.js, using require() and module.exports. Now, let's look at **ECMAScript Modules (ESM)**, which is the standardized module system for JavaScript and is gaining prominence in Node.js.

**The Evolution of JavaScript Modules**

Initially, JavaScript was primarily used for client-side web development (DOM manipulation, interactivity). There was no built-in module system because scripts were typically loaded sequentially, and global variables were often used (leading to conflicts, as discussed).

However, with the advent of Node.js, JavaScript began to be used for backend development. This meant handling complex server-side logic, managing many files, and avoiding global scope pollution became critical. Node.js introduced CommonJS to address these needs. Later, the JavaScript language itself evolved to include a native module system: ECMAScript Modules (ESM).

**Key Differences: CommonJS vs. ESM**

The primary difference lies in their syntax for importing and exporting:

|  |  |  |
| --- | --- | --- |
| Feature | CommonJS (CJS) | ECMAScript Modules (ESM) |
| Exporting | module.exports = ... or exports.name = ... | export const name = ..., export function name() { ... }, export default ... |
| Importing | require('./path/to/module') | import { name } from './path/to/module' or import defaultExport from './path/to/module' |
| File Extension | .js (by default) | .mjs (explicitly) or .js (with "type": "module" in package.json) |
| Execution | Synchronous loading | Asynchronous loading (static imports) |
| \_\_dirname/\_\_filename | Available globally | Not directly available; need import.meta.url |

Export to Sheets

Let's illustrate ESM with examples:

**ESM Exporting and Importing:**

**file2.mjs (or file2.js if type: "module" in package.json)**

// file2.mjs

export const a = 10;

export const b = 20;

export function add(param1, param2) {

return param1 + param2;

}

// You can also have a default export (only one per module)

const greeting = "Hello from ESM!";

export default greeting;

**file1.mjs (or file1.js if type: "module" in package.json)**

// file1.mjs

// Named Imports (must use curly braces and exact names)

import { a, b, add } from './file2.mjs';

// Default Import (no curly braces, can use any name)

import myGreeting from './file2.mjs';

console.log(`Named Imports: a = ${a}, b = ${b}, add(2, 3) = ${add(2, 3)}`);

console.log(`Default Import: ${myGreeting}`);

To run an ESM file in Node.js, you typically need to use the .mjs extension or add "type": "module" to your package.json. If you try to use require() in an .mjs file, Node.js will throw an error like require is not defined in ES module scope. Similarly, module.exports won't work in an ESM file; you must use export.

**Name Aliasing in ESM**

Aliasing in ESM is similar to CommonJS destructuring:

// file1.mjs

import { a as varA, add as addFunc } from './file2.mjs'; // Renaming during import

console.log(`Aliased: varA = ${varA}, addFunc(5, 5) = ${addFunc(5, 5)}`);

**Default vs. Named Exports**

* **Named Exports:** You can have multiple named exports from a single module. When importing, you must use the exact names (or alias them) and destructuring {}. This is good for exporting multiple related utilities.
* **Default Export:** You can have only one default export per module. When importing, you can give it any name you want, and you don't use {}. This is often used for the primary export of a module (e.g., a React component).

// myComponent.mjs

const MyComponent = () => { /\* ... \*/ };

export default MyComponent; // Default export

// utils.mjs

export const PI = 3.14; // Named export

export function sum(a, b) { return a + b; } // Named export

// app.mjs

import MainComponent from './myComponent.mjs'; // Any name for default

import { PI, sum } from './utils.mjs'; // Exact names for named

console.log(PI);

console.log(sum(1, 2));

// <MainComponent />

**Module Summary**

Node.js's module system is crucial for building maintainable and scalable applications. Here's a recap of what we've learned:

**Key Concepts:**

1. **Scope Isolation:** Each Node.js file is treated as a separate module, operating in its own isolated scope. This prevents global variable pollution and naming conflicts.
   * **Browser:** Variables declared with var can attach to the global window object.
   * **Node.js:** Variables are scoped to the module by default; require, module, exports, \_\_filename, and \_\_dirname are not global but are provided to each module.
2. **IIFE Module Wrapper:** Node.js wraps the content of every module file in an **Immediately Invoked Function Expression (IIFE)**. This IIFE takes exports, require, module, \_\_filename, and \_\_dirname as parameters, making them available within the module's scope without being truly global.

// Node.js internally wraps your code like this:

(function (exports, require, module, \_\_filename, \_\_dirname) {

// Your actual code from the .js file goes here

// Example:

// const myVar = 10;

// module.exports = myVar;

})(exports, require, module, \_\_filename, \_\_dirname);

1. **Types of Modules:**
   * **Local Modules:** Your own files, exported using module.exports (CJS) or export (ESM).
   * **Built-in Modules:** Core Node.js modules like fs, http, path, directly require()d or imported by name.
   * **Third-party Modules:** Packages installed via npm (e.g., express), also require()d or imported by name.
2. **CommonJS (CJS):**
   * Uses require() to import and module.exports to export.
   * Synchronous loading.
   * Default for .js files unless specified otherwise.
3. **ECMAScript Modules (ESM):**
   * Uses import and export keywords.
   * Asynchronous loading.
   * Requires .mjs file extension or "type": "module" in package.json.
   * Supports **Named Exports** (multiple items, require destructuring and exact names) and a single **Default Export** (can be imported with any name, no destructuring).
4. **Name Aliasing:** Both CJS and ESM allow renaming imported variables/functions to avoid conflicts or improve clarity (e.g., const { originalName: newName } = require(...) or import { originalName as newName } from ...).
5. **Index Export:** A common convention where an index.js (or index.mjs) file within a directory re-exports items from other files in that directory. This allows you to require() or import the directory directly (e.g., require('./utils')), making imports cleaner.

Understanding Node.js's module system is fundamental for writing robust, organized, and maintainable applications. While CommonJS is prevalent in many existing Node.js projects, ESM is the future and becoming increasingly common.

**Module 13: Explore Node.js Core Modules**

**Lecture 13-1: What is an event module?**

In this module, we'll explore Node.js's built-in modules, focusing on the highly important **events module**. As you know, Node.js operates on an **event-driven architecture**, and events play a crucial role in this paradigm.

**Understanding Events in Node.js**

At its core, Node.js's event-driven nature means it continuously listens for specific "events" or "actions" to occur. When an event happens (like a user making a request, data arriving, or a timer expiring), Node.js responds by executing a pre-defined **callback function**. This continuous listening and responding mechanism is powered by the **Event Loop**.

**The Event Loop in Action**

When you run a Node.js file, the **Event Loop** starts. Its primary job is to constantly **listen** for any events that are "emitted" or "triggered."

* If a **CPU-intensive task** comes in, the Event Loop will try to handle it within its single-threaded execution model.
* If an **I/O-intensive task** (like reading a file or making a network request) comes in, the Event Loop offloads it to the **worker pool** (or thread pool). Once the I/O operation is complete, a callback is placed back into the Event Loop's queue, and the Event Loop eventually executes that callback, sending a response back to the user.

This non-blocking, asynchronous behavior is a cornerstone of Node.js's efficiency, and it's all orchestrated by the Event Loop and the events it monitors.

**Event Emitters and Event Listeners**

For Node.js's event system to work, we need two main components:

1. **Event Emitter:** This is the entity responsible for **triggering** or **emitting** an event. Think of it as the source of an action.
2. **Event Listener:** This is the entity that **listens** for a specific event emitted by an Event Emitter. When the listened-for event occurs, the Event Listener executes a predefined **callback function** to handle the event.

This "emitter-listener" pattern forms the foundation of event-driven programming in Node.js.

**The events Module: EventEmitter Class**

Node.js provides a built-in events module, which contains the EventEmitter class. This EventEmitter class is fundamental for creating and managing custom events.

Let's illustrate this with an example. Imagine a school where a bell rings to signify the end of a class, and students react to it.

**Step 1: Importing EventEmitter**

First, we need to import the EventEmitter class from Node.js's built-in events module. We'll use the CommonJS syntax for this example, assuming our file has a .js extension.

// event.js

const EventEmitter = require('events'); // Correct way to import EventEmitter

// const EventEmitter = require('node:events'); // Also valid for built-in modules

**Step 2: Creating a Custom Emitter Class**

Now, we'll create our own class, say SchoolBell, and extend it from EventEmitter. This gives our SchoolBell class all the capabilities of an event emitter, including the ability to emit events and on (or addListener) to listen for them.

// event.js

const EventEmitter = require('events');

class SchoolBell extends EventEmitter {

constructor() {

super(); // Call the parent EventEmitter constructor

}

// This method will simulate the bell ringing and emitting an event

ring() {

console.log('School Bell: Ding! Dong! Ding!');

// Emit an event named 'bellRing' with a parameter 'ringSound'

this.emit('bellRing', 'ringSound');

}

}

In this SchoolBell class:

* The constructor calls super() to properly initialize the EventEmitter part of the class.
* The ring() method simulates the bell ringing. Importantly, it uses this.emit('bellRing', 'ringSound') to **trigger** an event named 'bellRing'. We're also passing a parameter, 'ringSound', along with the event.

**Step 3: Setting Up an Event Listener**

Now, let's create an instance of our SchoolBell and set up an **event listener** that reacts when the 'bellRing' event is emitted.

// event.js

const EventEmitter = require('events');

class SchoolBell extends EventEmitter {

constructor() {

super();

}

ring() {

console.log('School Bell: Ding! Dong! Ding!');

this.emit('bellRing', 'ringSound'); // Emitting the event

}

}

const schoolBell = new SchoolBell(); // Create an instance of our SchoolBell

// Set up an event listener for the 'bellRing' event

// The callback function will receive the 'ringSound' parameter

schoolBell.on('bellRing', (ringSound) => {

console.log(`Student: Yay! Class is over! Sound: ${ringSound}`);

});

// Trigger the event by calling the ring method

schoolBell.ring();

When you run node event.js, you'll see the following output:

School Bell: Ding! Dong! Ding!

Student: Yay! Class is over! Sound: ringSound

**Explanation:**

1. const schoolBell = new SchoolBell(); creates an instance of our custom emitter.
2. schoolBell.on('bellRing', (ringSound) => { ... }); registers a listener. It tells schoolBell: "When an event named 'bellRing' is emitted, execute this callback function, and whatever parameter was sent with the event (here, ringSound) will be passed to this callback."
3. schoolBell.ring(); calls the ring method, which in turn emits the 'bellRing' event.
4. Because the event was emitted, the registered listener's callback function executes, and the student's message is logged.

This example clearly demonstrates the core principle of Node.js's event-driven architecture: an event emitter triggers an event, and an event listener reacts to it by executing a specific callback function. This pattern is fundamental to how Node.js handles I/O, asynchronous operations, and many other functionalities.

**Lecture 13-2: Synchronous Way To Read And Write Files**

Let's dive into Node.js's **File System (FS) module** and learn how to read and write files synchronously.

**Understanding the File System Module**

The **File System (FS) module** in Node.js provides a rich set of functionalities for interacting with the file system on your machine. This includes:

* **Reading** data from existing files.
* **Creating** new files.
* **Writing** data to files.
* **Updating** and **Deleting** files.

Essentially, the FS module allows your Node.js application to perform standard file operations.

There are two primary ways to perform file operations in Node.js:

1. **Synchronous (blocking)**
2. **Asynchronous (non-blocking)**

Let's understand the key difference between these two approaches.

**Synchronous vs. Asynchronous File Operations**

The core of Node.js's power lies in its **asynchronous, non-blocking I/O model**, powered by the Event Loop and worker pool (or thread pool).

* **Asynchronous Way:**
  + When an **I/O-intensive task** (like reading a large file or a network request) comes in, the Node.js **Event Loop** offloads this task to the **worker pool** (a separate pool of threads).
  + The **single main thread (Event Loop) remains free** to process other incoming tasks.
  + Once the I/O task is completed by a worker thread, its callback function is pushed back into the Event Loop's queue.
  + The Event Loop then executes this callback, returning the result to your application.
  + This ensures that your application doesn't get "blocked" while waiting for slow I/O operations to complete, making Node.js highly performant for concurrent operations.
* **Synchronous Way:**
  + In contrast, when you perform a file operation **synchronously**, the Node.js **single main thread (Event Loop) itself handles the task**.
  + Crucially, **the Event Loop gets "blocked"** until that synchronous operation is fully completed.
  + This means that while a synchronous file read or write is in progress, no other incoming tasks (like other user requests or I/O operations) can be processed. They will have to **wait** until the synchronous operation finishes.
  + This can lead to significant performance bottlenecks and unresponsive applications, especially in server environments.

**When to use Synchronous Operations?** Generally, **you should avoid synchronous file operations in Node.js, especially in server-side code**, as they block the Event Loop. However, they might be acceptable for:

* **Startup tasks:** Reading configuration files at the very beginning of your application's lifecycle, before it starts serving requests.
* **Simple scripts:** Small, single-purpose scripts where blocking is not an issue.

**Synchronous File Reading with fs.readFileSync()**

Node.js's FS module provides synchronous versions of many file operations, typically identifiable by a Sync suffix (e.g., readFileSync, writeFileSync).

Let's demonstrate reading a file synchronously.

**1. Create a sample text file:** Let's create a file named hello.txt in your project directory with some content:

hello.txt

Hello World!

This is a test file.

**2. Implement Synchronous Read:** Now, in your Node.js file (e.g., fs.js), we'll read hello.txt synchronously.

// fs.js

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

console.log('Task 1: Starting file read...');

try {

// fs.readFileSync() reads the file synchronously

// Parameters: path, [options]

// The 'utf8' encoding is crucial to get readable text instead of a Buffer

const data = fs.readFileSync('./hello.txt', 'utf8');

console.log('Task 2: File content read successfully!');

console.log('File Content:', data);

} catch (error) {

console.error('Error reading file:', error.message);

}

console.log('Task 3: Finishing program...');

**Running the code (node fs.js):**

Task 1: Starting file read...

Task 2: File content read successfully!

File Content: Hello World!

This is a test file.

Task 3: Finishing program...

**Observation:** Notice the order of logs. "Task 2" (file content) appears *before* "Task 3". This is because fs.readFileSync() is a **blocking** operation. Node.js pauses its execution until the file read is complete, then it proceeds to the next line. If hello.txt were a very large file, "Task 3" would be delayed significantly.

**Important Note on Encoding:** When reading files, if you don't specify the encoding (e.g., 'utf8'), fs.readFileSync() will return a Buffer object (raw binary data). To get human-readable text, always specify the encoding, typically 'utf8'.

**Synchronous File Writing with fs.writeFileSync()**

Writing files synchronously follows a similar pattern using fs.writeFileSync().

**1. Implement Synchronous Write:** Let's modify fs.js to first write to hello.txt and then read it synchronously.

// fs.js

const fs = require('fs');

console.log('Task 1: Starting file operations...');

const newContent = 'Learning Node.js File System!';

try {

// fs.writeFileSync() writes to a file synchronously

// If the file exists, it will overwrite its content.

// Parameters: path, data (string or Buffer), [options]

fs.writeFileSync('./hello.txt', newContent);

console.log('Task 2: File written successfully!');

} catch (error) {

console.error('Error writing file:', error.message);

}

try {

// Read the updated content synchronously

const data = fs.readFileSync('./hello.txt', 'utf8');

console.log('Task 3: File content read successfully!');

console.log('Updated File Content:', data);

} catch (error) {

console.error('Error reading file after write:', error.message);

}

console.log('Task 4: All operations finished!');

**Running the code (node fs.js):**

Task 1: Starting file operations...

Task 2: File written successfully!

Task 3: File content read successfully!

Updated File Content: Learning Node.js File System!

Task 4: All operations finished!

**Observation:** Again, the synchronous nature is evident. "Task 2" completes before "Task 3" begins, and both complete before "Task 4". If either the write or read operation takes a long time, the entire program execution will pause until that specific operation is done.

This demonstrates how to perform synchronous file read and write operations in Node.js. While convenient for simple, sequential tasks, remember that these blocking operations can significantly impact performance in complex or server-side applications. We will explore the more recommended **asynchronous** approach in the next lecture.

**Lecture 13-3: Asynchronous Way To Read And Write Files**

In this lesson, we'll explore how Node.js's **File System (FS) module** handles operations in an **asynchronous (non-blocking)** manner. We've already seen the synchronous approach and understand that asynchronous operations are crucial for Node.js's efficiency. In the asynchronous model, file read/write tasks are offloaded to the **worker pool** via the Event Loop, allowing the main thread to remain unblocked and responsive.

**Asynchronous File Reading with fs.readFile()**

The asynchronous versions of FS module functions typically do not have a Sync suffix (e.g., fs.readFile, fs.writeFile). They follow the **error-first callback pattern**, which is a common convention in Node.js for asynchronous operations.

Let's demonstrate reading a file asynchronously.

**1. Create/Ensure hello.txt exists:** Make sure you have a hello.txt file in your project directory. Its content can be anything, for example:

Hello Async World!

**2. Implement Asynchronous Read:** Now, in your Node.js file (e.g., fs\_async.js), we'll read hello.txt asynchronously.

// fs\_async.js

const fs = require('fs');

console.log('Task 1: Starting asynchronous file read...');

// fs.readFile() reads the file asynchronously

// Parameters: path, [options], callback function

// The callback function has two parameters: (error, data)

fs.readFile('./hello.txt', 'utf8', (err, data) => {

// This callback executes ONLY when the file read operation is complete

if (err) {

console.error('Error reading file:', err.message);

return; // Important to return if there's an error

}

console.log('Task 2: File content read successfully (inside callback)!');

console.log('File Content:', data);

});

console.log('Task 3: Finishing program (this will print before file read completes)...');

**Running the code (node fs\_async.js):**

Task 1: Starting asynchronous file read...

Task 3: Finishing program (this will print before file read completes)...

Task 2: File content read successfully (inside callback)!

File Content: Hello Async World!

**Observation:** Notice the order of logs. "Task 3" prints *before* "Task 2" (which contains the file content). This clearly illustrates the **non-blocking** nature of fs.readFile(). Node.js offloads the file reading task, immediately moves on to console.log('Task 3: ...'), and only once the file is fully read does the callback function for "Task 2" execute.

**Error-First Callback Pattern:** The callback function for fs.readFile() (and most asynchronous Node.js functions) receives two arguments: err and data.

* If an error occurs during the operation (e.g., file not found), err will contain an Error object, and data will be undefined. You should always check for err first.
* If the operation is successful, err will be null, and data will contain the result (the file content in this case).

**Asynchronous File Writing with fs.writeFile()**

Writing files asynchronously follows the same error-first callback pattern using fs.writeFile().

**1. Implement Asynchronous Write:** Let's modify fs\_async.js to first write to a file asynchronously and then read it.

// fs\_async.js

const fs = require('fs');

console.log('Task 1: Starting asynchronous file operations...');

const newContent = 'This content was written asynchronously.';

// fs.writeFile() writes to a file asynchronously

// If the file exists, it will overwrite its head content.

// Parameters: path, data (string or Buffer), [options], callback function

fs.writeFile('./output.txt', newContent, 'utf8', (err) => {

// This callback executes when the file write operation is complete

if (err) {

console.error('Error writing file:', err.message);

return;

}

console.log('Task 2: File written successfully (inside write callback)!');

// After writing, let's read the content to verify

fs.readFile('./output.txt', 'utf8', (err, data) => {

if (err) {

console.error('Error reading file after write:', err.message);

return;

}

console.log('Task 3: File content read successfully (inside read callback)!');

console.log('Content from output.txt:', data);

});

});

console.log('Task 4: Main program execution continues (non-blocking)...');

**Running the code (node fs\_async.js):**

Task 1: Starting asynchronous file operations...

Task 4: Main program execution continues (non-blocking)...

Task 2: File written successfully (inside write callback)!

Task 3: File content read successfully (inside read callback)!

Content from output.txt: This content was written asynchronously.

**Observations:**

* "Task 4" prints before "Task 2" and "Task 3". This is because the fs.writeFile() operation is offloaded, and the main thread immediately proceeds to console.log('Task 4: ...').
* The fs.readFile() operation is nested inside the fs.writeFile() callback. This is a common pattern for dependent asynchronous operations, ensuring that the read operation only starts *after* the write operation has successfully completed. This nesting can lead to **"callback hell"** in very complex scenarios, which is why promises and async/await were introduced (we'll cover those later).
* For fs.writeFile(), the callback only receives an err parameter if an error occurred. There's no data parameter for successful write operations.

**Advantages of Asynchronous Operations**

* **Non-Blocking:** The most significant advantage. The Node.js Event Loop is not blocked, allowing your application to handle multiple requests concurrently. This is critical for building scalable and responsive web servers.
* **Efficiency:** By offloading I/O tasks to the worker pool, Node.js can maximize the utilization of system resources.
* **Responsiveness:** Users won't experience delays due to one slow operation.

Understanding asynchronous file operations is crucial for effective Node.js development. It allows you to leverage Node.js's strengths and build high-performance applications.

Next, we'll explore **Streams and Buffers**, which are efficient ways to handle large amounts of data in Node.js, particularly useful with file system operations.

**Lecture 13-4: Buffer and Streaming**

Let's explore the crucial Node.js concepts of **Streaming** and **Buffers**.

**What is Streaming?**

**Streaming** is a method for transferring data from one place to another. Instead of sending the entire data at once, streaming breaks the data into smaller **pieces** or **chunks**, sending these chunks as individual **packages**.

Think of it like ordering a pizza. Instead of getting the whole pizza at once, you might receive it slice by slice. Each slice is a piece of data. Or, consider how video streaming platforms like YouTube work. If you had to download an entire 2GB video file before watching it, it would take a long time. To solve this, YouTube streams the video. They divide the large video file into many smaller pieces, which are then sent to your device as packages. This ensures a smooth viewing experience.

When you play a video, you see a red playback line moving, and often a gray line below it fills up automatically. This gray line indicates the **buffered** data. Data continuously arrives from the server and is loaded into your device, allowing you to watch the video without interruptions. For example, while you're watching the first five minutes of a video, the data for the next five minutes is being loaded. This allows you to watch the video on demand, without the long wait of a full download, providing a seamless user experience.

**What is a Buffer?**

Each "package" or "piece" of data that's transferred during streaming is called a **Buffer**. A buffer is essentially a temporary storage area for a chunk of data.

**Benefits of Streaming and Buffers**

1. **Smooth User Experience:** Streaming provides a continuous and uninterrupted user experience, especially with large files. Users don't have to wait for the entire content to load.
2. **Efficient Memory Management:** Streaming and buffers are excellent for handling **memory shortages**. When you load an entire large file (e.g., a 2GB video), it needs to sit in your computer's RAM, which is temporary memory. While your SSD might be terabytes in size, your RAM is usually much smaller (e.g., 16GB, 32GB, or 64GB). Loading a huge file into limited RAM can cause your system to slow down or even hang.

With streaming, only a small chunk of data is loaded into memory at a time. As the user consumes that data, the memory is freed up, and the next chunk of data is loaded. This allows for efficient use of limited memory, preventing system slowdowns.

**Implementing Streams in Node.js**

Node.js's fs (File System) module provides functionalities for working with streams. We'll focus on **Readable Streams** (for reading data) and **Writable Streams** (for writing data).

Let's look at how to create and use these streams.

**1. Creating a Readable Stream**

We use fs.createReadStream() to create a readable stream from a file. This stream will emit data in chunks.

First, let's create a file named hello\_world.txt with some content. For demonstration, let's say it contains:

Hello World! This is a long text file for streaming demonstration.

Now, let's create a Node.js file (e.g., streams.js) to read from it:

// streams.js

const fs = require('fs');

// Create a readable stream

const readStream = fs.createReadStream('./hello\_world.txt', { encoding: 'utf8' });

// Listen for 'data' event: This event is emitted when a chunk of data is available.

readStream.on('data', (chunk) => {

console.log('Received chunk of data:');

console.log(chunk);

});

// Listen for 'end' event: This event is emitted when there's no more data to read.

readStream.on('end', () => {

console.log('--- Reading finished! ---');

});

// Listen for 'error' event: This event is emitted if an error occurs during reading.

readStream.on('error', (err) => {

console.error('Error during reading:', err.message);

});

When you run this code (node streams.js), you'll see the content of hello\_world.txt printed in chunks (if the file is large enough to be chunked), followed by "Reading finished!". If you try to read a non-existent file, the 'error' event will be triggered.

**2. Creating a Writable Stream**

Similarly, we use fs.createWriteStream() to create a writable stream to a file. We can then write data to this stream in chunks.

Let's modify our streams.js to also write to a new file:

// streams.js

const fs = require('fs');

// --- Readable Stream ---

const readStream = fs.createReadStream('./hello\_world.txt', { encoding: 'utf8' });

readStream.on('data', (chunk) => {

console.log('Received chunk of data (Reading):');

console.log(chunk);

});

readStream.on('end', () => {

console.log('--- Reading finished! ---');

});

readStream.on('error', (err) => {

console.error('Error during reading:', err.message);

});

// --- Writable Stream ---

// Create a writable stream to a new file named 'output.txt'

const writeStream = fs.createWriteStream('./output.txt', { encoding: 'utf8' });

const contentToWrite = 'This is new content written via a writable stream.';

// Write a chunk of data to the writable stream

writeStream.write(contentToWrite, (err) => {

if (err) {

console.error('Error writing data:', err.message);

} else {

console.log('Chunk written to output.txt');

}

});

// End the writable stream to signify that no more data will be written

writeStream.end(() => {

console.log('--- Writing finished and stream closed! ---');

});

writeStream.on('error', (err) => {

console.error('Error during writing:', err.message);

});

When you run this (node streams.js), output.txt will be created (or overwritten if it exists) with the content "This is new content written via a writable stream.". You'll see logs for both reading and writing operations. The writeStream.end() method is important as it signals that no more data will be written and the stream should be closed.

**Piping Streams**

One of the most powerful features of Node.js streams is **Piping**. Piping allows you to connect the output of a readable stream directly to the input of a writable stream. This is incredibly efficient for large file operations, as data flows directly without being fully buffered in memory.

Let's try piping our hello\_world.txt to copied\_hello.txt:

// streams\_pipe.js

const fs = require('fs');

console.log('Starting file copy using streams (Piping)...');

// Create a readable stream from the source file

const readStream = fs.createReadStream('./hello\_world.txt', { encoding: 'utf8' });

// Create a writable stream to the destination file

const writeStream = fs.createWriteStream('./copied\_hello.txt', { encoding: 'utf8' });

// Pipe the readable stream to the writable stream

readStream.pipe(writeStream);

// Listen for events to know when the operation is complete or if an error occurs

writeStream.on('finish', () => {

console.log('File successfully copied using piping!');

});

readStream.on('error', (err) => {

console.error('Error reading source file:', err.message);

});

writeStream.on('error', (err) => {

console.error('Error writing destination file:', err.message);

});

Run node streams\_pipe.js. You'll find a new file copied\_hello.txt with the exact content of hello\_world.txt. This single line readStream.pipe(writeStream); handles the entire data transfer efficiently, without consuming excessive memory.

Streams and Buffers are fundamental to handling I/O operations efficiently in Node.js, especially when dealing with large files or network data. They enable non-blocking, memory-efficient data processing, which is key to Node.js's performance.

Next, we'll delve into another interesting topic in Node.js.

**Lecture 13-5: Making A Basic Logger App & Path Module**

In this Lecture, we'll create a basic **Logger Application** using pure Node.js, and we'll interact with it through our terminal.

**Setting Up Our Logger Application**

Let's start by creating a new folder named logger-app in our module. Inside this folder, we'll create our main file, index.js.

/module-2

/logger-app

index.js

Open your terminal within the logger-app directory.

**Exploring Global Objects: process and \_\_dirname**

In previous modules, we discussed the **Immediately Invoked Function Expression (IIFE)** that wraps Node.js modules. This IIFE provides access to several global objects and variables without explicitly declaring them, such as module, exports, require, and importantly, \_\_dirname and process.

Let's start by exploring the process object.

// index.js

console.log(process);

When you run node index.js, you'll see a large object logged to your console. This process object contains various information about the current Node.js process, including:

* process.version: Node.js version.
* process.platform: Operating system platform (e.g., 'win32', 'linux', 'darwin').
* And many other details about the process's environment.

One particularly useful property within process for our logger application is process.argv.

**process.argv - Accessing Command Line Arguments**

process.argv is an array that contains the command-line arguments passed to the Node.js process when it was launched.

Let's log process.argv:

// index.js

console.log(process.argv);

If you run node index.js, you'll see an array with at least two elements:

1. The path to the Node.js executable.
2. The path to your index.js file.

Now, let's pass some arguments to our script:

// in Bash

node index.js Hello World I am learning Node.js

If you run this, process.argv will now show:

1. Path to Node.js executable.
2. Path to index.js.
3. 'Hello'
4. 'World'
5. 'I'
6. 'am'
7. 'learning'
8. 'Node.js'

Notice that each word is treated as a separate argument. We want to combine these words into a single log message. We can achieve this by slicing the process.argv array to remove the first two elements (the executable and file path) and then join() the remaining elements with a space.

// index.js

const messageArgs = process.argv.slice(2); // Get arguments starting from the third one

const text = messageArgs.join(' '); // Join them into a single string

console.log(text);

Now, if you run node index.js Hello World I am learning Node.js, the output will be:

Hello World I am learning Node.js

This is our desired log message!

**Basic Input Validation and Exit**

Before we start logging, we should ensure the user provides a message to log. If no message is provided, we'll inform the user and gracefully exit the application. We can use process.exit() for this.

// index.js

const messageArgs = process.argv.slice(2);

const text = messageArgs.join(' ');

if (!text) {

console.log('❌ Please provide a message to log.');

console.log('Example: node index.js Hello World');

process.exit(1); // Exit with a non-zero code to indicate an error

}

console.log(`Message to log: "${text}"`);

If you run node index.js without any arguments, it will print the error message and exit. If you provide arguments, it will log the message as before.

**Defining the Log File Path with path.join()**

Our logger will write messages to a log.txt file. We need a reliable way to define the path to this file, regardless of where our script is run. The \_\_dirname global variable (which gives the directory name of the current module) combined with the path module is perfect for this.

The path module is a built-in Node.js module that provides utilities for working with file and directory paths. Specifically, path.join() is useful for concatenating path segments in a platform-agnostic way (it handles forward slashes vs. backslashes correctly).

First, import the path module:

// index.js

const path = require('path'); // Import the path module

const messageArgs = process.argv.slice(2);

const text = messageArgs.join(' ');

if (!text) {

console.log('❌ Please provide a message to log.');

console.log('Example: node index.js Hello World');

process.exit(1);

}

// Define the log file path

const filePath = path.join(\_\_dirname, 'log.txt');

console.log(`Log file path: ${filePath}`);

If you run this, you'll see the full, correctly formatted path to log.txt within your logger-app directory.

**Appending Messages to the Log File with fs.appendFile()**

Now, let's append our text message to the log.txt file. We'll use the fs (File System) module, specifically fs.appendFile(), which is an asynchronous function to append data to a file.

First, import the fs module:

// index.js

const fs = require('fs'); // Import the fs module

const path = require('path');

const messageArgs = process.argv.slice(2);

const text = messageArgs.join(' ');

if (!text) {

console.log('❌ Please provide a message to log.');

console.log('Example: node index.js Hello World');

process.exit(1);

}

const filePath = path.join(\_\_dirname, 'log.txt');

// Append the message to the log file

fs.appendFile(filePath, text, 'utf8', (err) => {

if (err) {

console.error('Error logging message:', err.message);

return;

}

console.log('✅ Log added successfully!');

});

Now, if you run node index.js Hello World, you'll see "Log added successfully!". If you check your log.txt file, "Hello World" will be appended to it.

**Adding Timestamps and Newlines**

To make our logger more useful, let's add a timestamp to each log entry and ensure each log appears on a new line.

We'll use a **template literal** (backticks `) for easier string concatenation and to include newlines.

// index.js

const fs = require('fs');

const path = require('path');

const messageArgs = process.argv.slice(2);

const text = messageArgs.join(' ');

if (!text) {

console.log('❌ Please provide a message to log.');

console.log('Example: node index.js Hello World');

process.exit(1);

}

const filePath = path.join(\_\_dirname, 'log.txt');

// Get current timestamp

const timestamp = new Date().toISOString(); // e.g., '2025-06-03T19:30:00.000Z'

// Format the log message with timestamp and a newline

const logMessage = `[${timestamp}] ${text}\n`;

fs.appendFile(filePath, logMessage, 'utf8', (err) => {

if (err) {

console.error('Error logging message:', err.message);

return;

}

console.log('✅ Log added successfully!');

});

Now, if you run the script multiple times with different messages:

// in Bash

node index.js First log entry

node index.js This is a second message

Your log.txt file will look something like this:

[2025-06-03T19:33:48.000Z] First log entry

[2025-06-03T19:33:55.000Z] This is a second message

**Summary**

In this video, we built a basic command-line logger application using Node.js. We learned about:

* The **process.argv** array for accessing command-line arguments.
* The **path.join()** method from the path module for creating platform-independent file paths.
* The **fs.appendFile()** method for asynchronously appending data to a file.
* Using **timestamps** and **newlines** for better log formatting.

This basic logger can be a starting point for more complex logging solutions.

Next, we'll dive into building an **HTTP server** with Node.js and understand how concepts like localhost:3000 or localhost:5000 work!

**Lecture 13-1: How the Web Works**

When we open a website

**Q57) What is the sessionStorage API in JavaScript?**

**Answer:** sessionStorage is similar to localStorage, but with one key difference: it only persists data for the duration of the page session. A session ends when the browser or tab is closed. It is typically used for storing temporary data that only needs to be available during a single session.**Example:**

sessionStorage.setItem('theme', 'dark');

let theme = sessionStorage.getItem('theme');

console.log(theme); // Outputs "dark"

In this example, the setItem() method stores a value in the session storage, and the getItem() method retrieves it. The data will be cleared when the session ends (i.e., when the tab is closed).

**Q58) What is the Geolocation API in JavaScript?**

**Answer:** The Geolocation API allows websites to access the geographical location of a user's device. It is commonly used for applications like maps or location-based services. The API provides methods to get the user's current position or watch for changes in their position..**Example:**

navigator.geolocation.getCurrentPosition(function(position) {

console.log('Latitude: ' + position.coords.latitude);

console.log('Longitude: ' + position.coords.longitude);

});

In this example, getCurrentPosition() is used to retrieve the user's current geographical coordinates. The position object contains the coords property with latitude and longitude.

**Q59) How does the Web Storage API differ from cookies in JavaScript?**

Answer: The Web Storage API (which includes localStorage and sessionStorage) differs from cookies in several ways:

* Storage Size: Web Storage can store larger amounts of data (up to 5-10MB per domain) compared to cookies, which are limited to around 4KB.
* Lifetime: localStorage persists data across sessions, while cookies can have expiration dates set by the server.
* Data Handling: Cookies are sent with every HTTP request, while data in Web Storage is stored on the client side and not transmitted with requests, improving performance.
* Simplicity: Web Storage is easier to use and more efficient for client-side storage compared to cookies.

**Q60) What is the Notification API in JavaScript?**

**Answer:** The Notification API allows web pages to display notifications to the user, even if the page is not in the foreground. This API is often used in conjunction with service workers to send push notifications for real-time updates, such as messages or alerts...**Example:**

if (Notification.permission === 'granted') {

new Notification('Hello, you have a new message!');

} else {

Notification.requestPermission().then(permission => {

if (permission === 'granted') {

new Notification('Hello, you have a new message!');

}

});

}

In this example, the Notification object is used to display a notification. Before sending a notification, the browser must request permission to show notifications.

📝 Common Interview Follow-Ups(will be added later):

* How would you handle errors in the Fetch API?
* What are some common use cases for the localStorage and sessionStorage APIs?
* How do you ensure compatibility for the Geolocation API across different browsers?
* How do you handle JSON responses from APIs using the Fetch API?
* What are some security concerns when using localStorage and sessionStorage?
* How can you set expiration for localStorage data?
* How would you handle sending data through the Fetch API using POST requests?
* How can you use the Notification API to send push notifications with service workers?