**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.

**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?