**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.

**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?