Day 1: Introduction to Node.js

- Overview of Node.js and its uses
- Installation and setup
- Running your first Node.js script
- Understanding Node.js architecture (Event Loop, Single-threaded)

Overview of Node.js and its Uses

- **What is Node.js?**
  - Node.js is an open-source, cross-platform runtime environment for executing JavaScript code outside of a browser.
  - It was developed by Ryan Dahl in 2009 and allows developers to use JavaScript on the server side.
- **Key Features of Node.js:**
  - **Asynchronous and Event-Driven:** Handles requests without waiting for responses, making it suitable for handling many connections at once.
  - **Non-blocking I/O Model:** Efficiently manages input/output operations, enhancing performance for real-time applications.
  - **Single-Threaded Architecture:** Uses a single main thread with an event loop, which we'll explore more deeply later.
- **Common Uses of Node.js:**
  - **Web Applications and REST APIs:** Ideal for building fast, scalable web servers.
  - **Real-Time Applications:** Chat applications, gaming servers, collaborative tools (like Google Docs).
  - **Microservices:** Node's modularity makes it popular for microservice architectures.
  - **Tools and Scripts:** Often used to automate tasks, manage file systems, and run development tools.

---

2. Installation and Setup

- **Step-by-Step Installation:**
  - **Download Node.js:**
    - Visit Node.js Official Website.
    - Choose the recommended version (LTS) for stability or the latest version for newer features.
  - **Installation Process:**
    - Run the downloaded installer and follow the instructions.
    - The installer will also install `npm` (Node Package Manager), which is used to install libraries and manage dependencies.
  - **Verify Installation:**
    - Open a terminal (or command prompt) and check if Node.js and npm are installed by running:
      - `node -v`
      - `npm -v`

Running Your First Node.js Script

- **Creating a Simple Script:**
  - Open a text editor and create a new file named `app.js`.
  - Write the following code in `app.js`:

```
console.log("Event 1")
```

Running the Script:

- In the terminal, navigate to the folder where you saved `app.js`.
- Run the script using the following command

node app.js

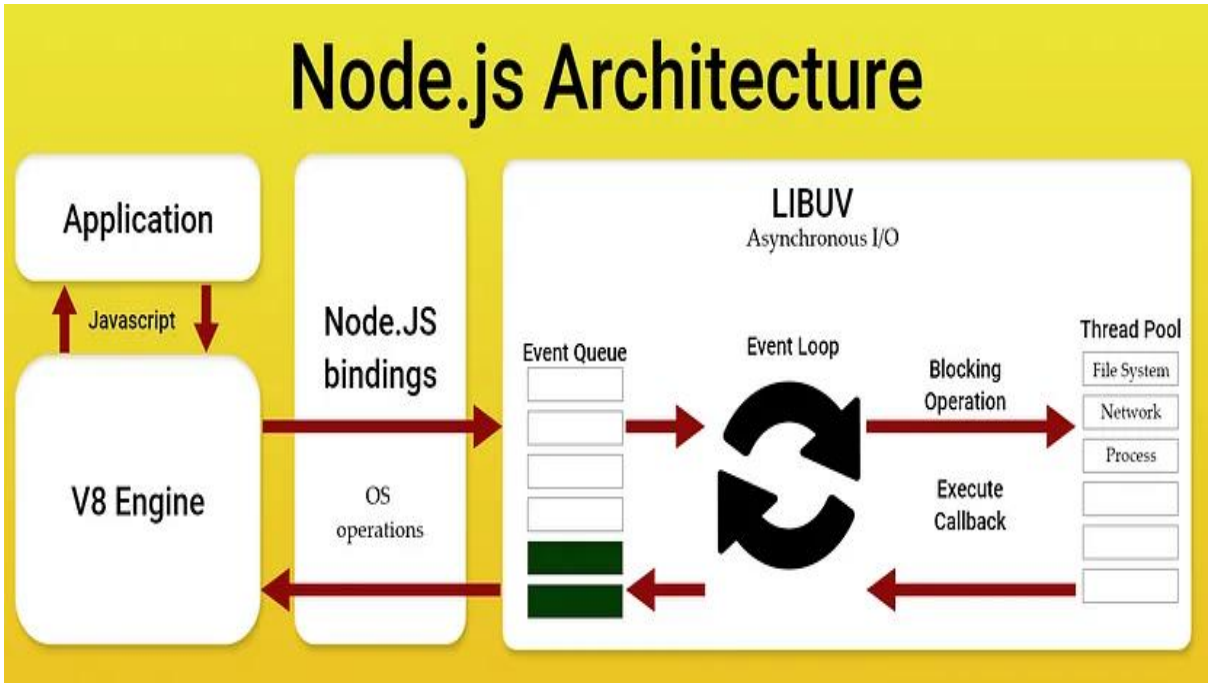
Understanding Node.js Architecture (Event Loop, Single-Threaded)

- **Single-Threaded Architecture:**
  - Unlike many server environments, Node.js operates on a single main thread.
  - Rather than creating a new thread for each request, Node.js uses a single thread to handle multiple requests concurrently.
  - This is made possible by its event-driven, non-blocking architecture.
- **Event Loop:**
  - The **event loop** is central to how Node.js handles asynchronous operations.
  - When a request is made, the event loop handles it, sending any tasks that might take time (e.g., reading a file or making a database request) to the system's background.
  - When the background operation completes, the event loop picks up the response and completes the request.
  - This approach allows Node.js to handle thousands of connections at once without creating multiple threads, making it highly efficient for I/O-bound tasks.

```
console.log("Event 1")


setTimeout(function(){
    console.log("Event 2")
},5000)

console.log("Event 3")
```



V8 Engine (JavaScript Execution)

- **What it is:** The **V8 Engine** is a powerful component that converts JavaScript code into machine code. This engine is responsible for executing the JavaScript code within Node.js.
- **How to explain:** It's like a translator that converts JavaScript (which is easy for us to understand) into instructions that the computer can execute.
- **Key Point:** Whenever your Node.js application runs JavaScript, it gets processed by the V8 engine.

V8 engine vs Bubble:

"JSX is converted to JavaScript using tools like Babel or the React environment, and this JavaScript code is then executed by the V8 engine, which converts it into machine code that the computer can understand."

Node.js Bindings

- **What it is: Node.js bindings** are the connectors that allow JavaScript (running in V8) to communicate with lower-level system operations like reading from the file system or making network requests.
- **How to explain:** The bindings are like the **tools** that help the JavaScript engine communicate with the **system** (e.g., your computer or server). It helps the app access things like files or networks.

Event Loop

- **What it is:** The **Event Loop** is the core of Node.js, handling all incoming tasks (like client requests) efficiently.

- **How to explain:** The event loop works like a **manager** in a restaurant. When tasks (like customer orders) come in, it assigns them to workers (background tasks) and continues to accept more orders without waiting.
- **Key Point:** The event loop doesn't handle the long-running tasks itself (e.g., reading a file), but it manages them and continues to accept new requests. Once a task is done, it comes back to the event loop.

Event Queue

- **What it is:** The **Event Queue** holds all the tasks that need to be processed by the event loop.
- **How to explain:** Think of it as a **waiting area** where tasks wait until the event loop picks them up for processing.
- **Key Point:** All incoming tasks are stored in the event queue before being handled by the event loop.

Blocking Operation & Execute Callback

- **Blocking Operation:** Refers to tasks like accessing the file system or querying a database, which takes time to complete. Instead of blocking the whole application, these tasks are given to LIBUV to handle in the background.
- **Execute Callback:** Once LIBUV completes the task, it notifies the event loop with a **callback** (which is like a message saying "task done").

Interview Questions and Answers:

What is the JavaScript runtime environment?

Answer:

- The **JavaScript runtime environment** is where JavaScript code is executed. It includes components like the **Call Stack**, **Web APIs**, **Task Queue**, **Microtask Queue**, and the **Event Loop**.
- **Call Stack:** Keeps track of function execution in a Last-In-First-Out (LIFO) order.
- **Web APIs:** Provide browser-specific features like setTimeout, fetch, localStorage, etc.
- **Task Queue:** Holds asynchronous tasks (like setTimeout) that are ready to be processed after the current stack is clear.
- **Microtask Queue:** Holds promises and other microtasks, processed before the Task Queue.
- **Event Loop:** Continuously checks if the call stack is empty and if there are tasks in the queues ready to be executed.

Can you explain how the JavaScript Event Loop works?

Answer:

- The **Event Loop** ensures that JavaScript executes code in a non-blocking way, even though it's single-threaded.

- When JavaScript runs, tasks are pushed to the **Call Stack**. If the call stack is busy (e.g., running functions), new tasks are queued in the **Task Queue** or **Microtask Queue**.
- The **Event Loop** checks the **Call Stack** for any pending tasks. If the stack is empty, it checks the **Microtask Queue** first (e.g., resolved promises) and processes them. If the Microtask Queue is also empty, the **Task Queue** is processed (e.g., setTimeout callbacks).

```
console.log('Start');

setTimeout(() => {
  console.log('Timeout');
}, 0);

Promise.resolve().then(() => {
  console.log('Promise');
});

console.log('End');
```

Output:

- Start
- End
- Promise (from Microtask Queue)
- Timeout (from Task Queue)

What is the difference between the Call Stack and the Task Queue?

Answer:

- The **Call Stack** is where JavaScript functions are executed. It follows the Last-In-First-Out (LIFO) principle, meaning the most recently added function gets executed first.
- The **Task Queue** holds asynchronous tasks (e.g., setTimeout or setInterval) that are waiting to be executed once the **Call Stack** is empty.

Example:

- A function like setTimeout places its callback in the **Task Queue** after the specified delay. The callback will only be executed after the **Call Stack** has cleared.

What are Web APIs in JavaScript, and why are they important?

Answer:

- **Web APIs** are browser-provided functionalities that extend the capabilities of JavaScript beyond the core language. They include things like fetch, setTimeout, localStorage, DOM manipulation, and more.

- These APIs allow JavaScript to interact with the browser and perform tasks like making network requests, manipulating the DOM, or handling timers.

What is the difference between the Microtask Queue and the Task Queue?

Answer:

- The **Microtask Queue** holds tasks like resolved promises and other microtasks. The **Task Queue** holds regular asynchronous tasks like setTimeout or setInterval callbacks.
- **Microtasks** (e.g., resolved promises) are given higher priority and are executed immediately after the current execution context (or after the **Call Stack** is empty) and before any tasks in the **Task Queue**.
- **Tasks** in the **Task Queue** are processed after the **Call Stack** and **Microtask Queue** are empty.

```
console.log('Start');

setTimeout(() => {
  console.log('Task Queue');
}, 0);

Promise.resolve().then(() => {
  console.log('Microtask Queue');
});

console.log('End');
```

- Start
- End
- Microtask Queue (processed before the Task Queue)
- Task Queue

What happens when you run setTimeout() with a delay of 0 milliseconds?

Answer:

- The callback passed to setTimeout with a delay of 0 milliseconds is not executed immediately. Instead, it is placed in the **Task Queue**, and will only be executed after the current **Call Stack** is cleared and all **Microtasks** are processed.

```
setTimeout(() => {
  console.log('Task Queue (0 ms)');
}, 0);

console.log('Main thread work');
```

- Main thread work
- Task Queue (0 ms)