Great 👍 Let's move on to **2. Node.js Architecture**.
Since this is one of the **most asked interview topics**, I'll break it down in **clear layers** with **developer-friendly notes** and **Q&A points**.

---

2. Node.js Architecture

📌 Overview

- Node.js is **single-threaded** but can handle **thousands of concurrent requests** due to its **event-driven, non-blocking I/O architecture**.
- At its core:
    1. **V8 Engine** → Executes JavaScript.
    2. **libuv** → Handles async I/O operations.
    3. **Event Loop** → Manages request callbacks.

📌 Components of Node.js Architecture

1. **V8 Engine**
    - Developed by Google (used in Chrome).
    - Converts JavaScript → machine code.
    - Extremely fast due to Just-In-Time (JIT) compilation.
2. **Single Thread**
    - Node.js uses **one main thread** to handle requests.
    - Instead of blocking, it **delegates heavy I/O work** to background workers (via libuv).
3. **libuv**
    - A C++ library that provides **event loop + thread pool**.
    - Handles async tasks (file system, DNS, network, crypto).
    - Uses a **thread pool** (default: 4 threads) for heavy I/O tasks.
4. **Event Loop**
    - The "heart" of Node.js.
    - Continuously checks the **callback queue** and executes pending tasks.
    - Allows **non-blocking async execution**.
5. **Callback Queue**
    - Stores functions (callbacks) that need to be executed.
    - Example: After DB query finishes → its callback goes into queue → event loop picks it up.
6. **APIs / Async Operations**

- Node.js delegates operations (like DB queries, file read/write) to **libuv**.
- Once done → results are pushed to **callback queue** → processed by event loop.

📌 Node.js Request Lifecycle

1. Client sends a request (e.g., DB query, file read).
2. Node.js receives the request in **single thread**.
3. If **non-blocking I/O** → task is offloaded to **libuv** (background thread).
4. Once completed → callback placed in **callback queue**.
5. **Event loop** picks it and executes callback in main thread.

👉 This makes Node.js **asynchronous & scalable** without creating multiple threads for each request.

📌 Example: File Read

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

console.log("Start");

// Non-blocking async read
fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log("File Content:", data);
});

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

Output (order matters):

```
Start
End
File Content: Hello World
```

👉 Because `fs.readFile` is async → offloaded to libuv → event loop executes callback later.

📌 Where Multi-Threading Comes In

- Node.js itself is **single-threaded**.
- But **libuv's thread pool** handles expensive operations (like file I/O, DNS, crypto).
- For **CPU-intensive work** → use:
  - **Worker Threads** (since Node 10.5+)
  - **Clustering** (multiple Node.js processes on multi-core CPU)

📌 Interview Q&A

☑ **Q: Is Node.js single-threaded?**
👉 Yes, Node.js runs JavaScript in a single thread, but uses **libuv's thread pool** for async tasks.

☑ **Q: What is the Event Loop?**
👉 A loop that continuously checks for pending callbacks/events and executes them → enables non-blocking I/O.

☑ **Q: How does Node.js handle multiple requests concurrently?**
👉 By offloading blocking tasks to **libuv thread pool**, while event loop continues processing other requests.

☑ **Q: Difference between process.nextTick() and setImmediate()?**
👉 `process.nextTick()` runs **before next event loop iteration**; `setImmediate()` runs in the **next iteration** of event loop.

☑ **Q: How to handle CPU-heavy tasks in Node.js?**
👉 Use **Worker Threads** or **Cluster Mode** to distribute load.

📌 Key Takeaways

- Node.js = **single-threaded event-driven model**.
- Uses **libuv** for async I/O and background thread pool.
- **Event Loop** ensures non-blocking concurrency.
- Heavy CPU tasks → offload to **workers/clusters**.

👉 Next, we can move to **3. Core Modules in Node.js** (fs, http, path, os, events, stream).

Do you want me to explain **all core modules together** in one go, or should I break them into **separate small sections** (like `fs`, `http`, `stream` etc.)?