Perfect. Continuing **systematically**, **NotebookLM-ready**, with **NOTES and CODE strictly separated**.

📘 TOPIC: Browser vs Node.js Runtime

(Same JavaScript • Different Capabilities • Internals • Interview Traps)

🟦 NOTE 1: JavaScript vs Runtime (core distinction)

- **JavaScript** = the language (ECMAScript)
- **Runtime** = environment that runs JS
- Runtime provides:
  - Global object
  - APIs
  - Event loop integration
  - I/O capabilities

👉 Same JS code, **different powers** depending on runtime.

🟦 NOTE 2: What browsers provide

Browser runtime includes:

- DOM
- BOM (`window`, `location`, `history`)
- Web APIs:
  - `setTimeout`
  - `fetch`
  - `localStorage`
  - Events
- Rendering engine

Browser JS is **UI-focused**.

🟩 CODE 1: Browser-only globals

```js

window;
document;
localStorage;
alert;
```

🟦 NOTE 3: What Node.js provides

Node.js runtime includes:

- File system access

- Network access
- OS-level APIs
- `require` / `import`
- `process`
- `Buffer`
- No DOM

Node JS is **server / system-focused**.

🟩 CODE 2: Node-only globals

```
js

global;
process;
__dirname;
__filename;
Buffer;
```

🟦 NOTE 4: Global object differences

Aspect	Browser	Node
Global object	`window`	`global`
Standard access	`globalThis`	`globalThis`
DOM access	Yes	No

🟩 CODE 3: Global access

```
js

globalThis === window; // Browser: true
globalThis === global; // Node: true
```

🟦 NOTE 5: Module systems

Browser

- ES Modules only
- `<script type="module">`
- Uses `import` / `export`

Node.js

- CommonJS (`require`)
- ES Modules (`import`)
- Depends on config

🟩 CODE 4: Browser module

```js
// browser.js
export const x = 10;
```

```html
<script type="module" src="browser.js"></script>
```

🟩 CODE 5: Node CommonJS

```js
const fs = require("fs");
```

🟦 NOTE 6: File system access

- Browser: ❌ (sandboxed)
- Node: ✅ full access

🟩 CODE 6: Node file system

```js
const fs = require("fs");

fs.readFileSync("data.txt", "utf8");
```

🟦 NOTE 7: Networking

- Browser:
  - `fetch`
  - Subject to CORS
- Node:
  - `http` , `https`
  - No CORS restriction (server-side)

🟩 CODE 7: Fetch comparison

```js
```

```
// Browser & modern Node
fetch("/api/data");
```

🟦 NOTE 8: Event loop differences (important)

Browser

- Single event loop
- Rendering between tasks
- UI blocking is critical

Node.js

- libuv-based loop
- Multiple phases
- No rendering

🟩 CODE 8: Node event loop phases (conceptual)

```
txt

Timers → I/O → Poll → Check → Close
```

🟦 NOTE 9: Timers behavior differences

- `setTimeout` exists in both
- Precision and scheduling differ
- Node timers are less tied to UI

🟩 CODE 9: Timer example

```
js

setTimeout(() => {
  console.log("Runs later");
}, 0);
```

🟦 NOTE 10: Security model

Browser

- Strong sandbox
- No direct OS access
- Same-origin policy

Node

- Full system access

- Security depends on developer

🟦 NOTE 11: Performance considerations

- Browser:
  - JS + rendering compete
  - Heavy JS hurts UX
- Node:
  - JS handles many clients
  - Blocking code kills throughput

🟩 CODE 10: Blocking danger in Node

```js
while (true) {}
// Freezes entire server
```

🟦 NOTE 12: Use cases

Browser

- UI
- Interactions
- Animations
- Client-side logic

Node.js

- APIs
- Databases
- File handling
- Real-time servers

🟦 NOTE 13: Common interview traps

- JavaScript ≠ Browser
- Node is not multithreaded JS
- DOM is not part of JS
- Event Loop exists in both, but differs
- `window` is undefined in Node

🟦 FINAL MENTAL MODEL

- JavaScript is universal
- Runtime defines capabilities
- Browser = UI + Web APIs

- Node = OS + Server APIs
- Same syntax, different powers
- `globalThis` unifies globals

🛑 **STOP HERE**

Next **recommended continuation**:

1️⃣ **Fetch API (deep, real-world async flow)**
2️⃣ **Error Handling (sync vs async, try/catch pitfalls)**
3️⃣ **Memory Management & Garbage Collection**

Reply with **next** or the **number** you want to continue.