Perfect 🔥 — let's move on to **12. Streams in Node.js**.
Streams are one of the **most powerful concepts in Node.js**, built on top of the **EventEmitter**, and they are **heavily used in production systems** (files, network, video/audio, logs).

---

12. Streams in Node.js

📌 What are Streams?

- **Definition**: Streams are objects that let you **read or write data sequentially (piece by piece, chunk by chunk)** instead of loading the whole data into memory.
- They make **I/O operations efficient** (especially with large files, videos, or real-time data).

👉 Example:
Instead of loading a 2 GB file into memory, you **read it in chunks** and send each chunk to the client.

📌 Why Streams?

- **Memory efficient** → handles data in chunks.
- **Faster processing** → data can be processed while still being read/written.
- **Scalable** → avoids blocking the event loop with large data.

📌 Types of Streams

1. **Readable** → can read data (e.g., `fs.createReadStream()`).
2. **Writable** → can write data (e.g., `fs.createWriteStream()`).
3. **Duplex** → both readable & writable (e.g., TCP sockets).
4. **Transform** → duplex streams that modify data while passing through (e.g., compression, encryption).

📌 Stream Events

Since streams extend **EventEmitter**, they emit events like:

- `data` → when a chunk is available.
- `end` → when no more data.
- `error` → when an error occurs.
- `finish` → when all data is written.

📌 Examples

1. Readable Stream

```
const fs = require('fs');
const readStream = fs.createReadStream('bigfile.txt', { encoding: 'utf8' });

readStream.on('data', chunk => {
  console.log('Received chunk:', chunk.length);
});

readStream.on('end', () => {
  console.log('File reading completed');
});
```

👉 Reads file **chunk by chunk** instead of all at once.

---

2. Writable Stream

```
const fs = require('fs');
const writeStream = fs.createWriteStream('output.txt');

writeStream.write('Hello, ');
writeStream.write('World!\n');
writeStream.end();

writeStream.on('finish', () => {
  console.log('Data written successfully');
});
```

---

3. Pipe (Readable → Writable)

This is the **real power of streams**.

```
const fs = require('fs');
const readStream = fs.createReadStream('input.txt');
const writeStream = fs.createWriteStream('output.txt');

readStream.pipe(writeStream);
```

👉 `pipe()` automatically handles backpressure.

---

4. Duplex Stream (TCP Example)

```
const net = require('net');
const server = net.createServer((socket) => {
  socket.write('Hello! You are connected.\n');
  socket.on('data', (data) => {
    console.log('Client says:', data.toString());
  });
});

server.listen(4000, () => {
  console.log('Server running on port 4000');
});
```

---

5. Transform Stream (Compression Example)

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

const readStream = fs.createReadStream('input.txt');
const writeStream = fs.createWriteStream('input.txt.gz');
const gzip = zlib.createGzip();

readStream.pipe(gzip).pipe(writeStream);

console.log('File compressed successfully');
```

👉 `zlib` transforms data while streaming.

📌 Backpressure in Streams

- Happens when **data is being pushed faster than it can be consumed**.
- `pipe()` handles backpressure automatically.
- Without `pipe()`, you must use `stream.write()` with `drain` event handling.

Example:

```
if (!writeStream.write(chunk)) {
  readStream.pause();
  writeStream.once('drain', () => readStream.resume());
}
```

📌 Real-World Use Cases

1. **File handling** → large file read/write.
2. **Video/Audio streaming** → Netflix, YouTube-like apps.

3. **Log processing** → reading/writing logs in chunks.
4. **Compression & Encryption** → `zlib`, `crypto`.
5. **Network sockets** → chat apps, real-time APIs.

📌 Interview Questions

1. **What are streams in Node.js? Why are they useful?**
   → Streams process data chunk by chunk, reducing memory usage and increasing performance.

2. **Explain different types of streams.**
   → Readable, Writable, Duplex, Transform.

3. **What is backpressure in streams? How do you handle it?**
   → When writable cannot consume as fast as readable produces. Handled via `pipe()` or `drain` event.

4. **How is a Transform stream different from a Duplex stream?**
   → Both can read & write, but Transform modifies data in the process.

5. **Give real-world examples of Node.js streams.**
   → File I/O, video streaming, chat apps (TCP), gzip compression.

☑ **Key Takeaway**:

- Streams are **memory-efficient & scalable** for handling large data.
- Use `pipe()` for simple cases (automatic backpressure).
- Understand **backpressure, events, and Transform streams** for advanced use.

👉 Next big topic is **13. Node.js Clustering & Scaling** (essential for performance & load balancing in multi-core CPUs).

Do you want me to go into **Clustering & Worker Threads** next, or cover **Buffer & Binary Data Handling** before scaling?