Perfect, let's dive into **13. Node.js Clustering & Scaling** 🚀

13. Node.js Clustering & Scaling

📌 Why Clustering & Scaling?

- Node.js is **single-threaded** → by default, it runs on **one CPU core**.
- Modern servers have **multiple cores**, so one Node process can't utilize all resources.
- **Clustering & Scaling** allow Node apps to handle **more load, better concurrency, and high availability**.

1. Node.js Cluster Module

- **Cluster module** allows you to **fork multiple worker processes** that share the same server port.
- Each worker = separate Node.js instance, running on different CPU cores.
- **Master process** manages workers and load balances requests.

Example:

```
const cluster = require("cluster");
const http = require("http");
const os = require("os");

if (cluster.isMaster) {
  const numCPUs = os.cpus().length;
  console.log(`Master ${process.pid} is running`);

  // Fork workers
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  // Restart worker if it crashes
  cluster.on("exit", (worker) => {
    console.log(`Worker ${worker.process.pid} died, restarting...`);
    cluster.fork();
  });

} else {
  // Workers share TCP connection
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end(`Hello from Worker ${process.pid}`);
  }).listen(3000);
```

```
  console.log(`Worker ${process.pid} started`);
}
```

👉 Each request is distributed among workers.
👉 If one worker crashes, others keep working.

---

2. Scaling Node.js Applications

a) Vertical Scaling

- Add more **CPU, RAM** to a single machine.
- Limited by hardware → doesn't fully solve scalability.

b) Horizontal Scaling

- Run Node.js across **multiple machines**.
- Use **load balancers** (e.g., **NGINX, HAProxy, AWS ELB**) to distribute requests.
- Best for **large-scale production apps**.

---

3. Load Balancing

- **Built-in Cluster module** does round-robin load balancing (on most OS).
- **NGINX / PM2** often used for more control.

Example: **NGINX Load Balancer Config**

```
upstream node_app {
    server 127.0.0.1:3000;
    server 127.0.0.1:3001;
    server 127.0.0.1:3002;
}

server {
    listen 80;
    location / {
        proxy_pass http://node_app;
    }
}
```

---

4. PM2 for Process Management

- **PM2** = Production-ready process manager for Node.js.
- Handles **clustering, monitoring, auto-restart, logging**.

Commands:

```
# Start app with cluster mode using all CPU cores
pm2 start app.js -i max

# Monitor processes
pm2 monit

# Restart all processes
pm2 restart all

# Generate startup script (auto start on reboot)
pm2 startup
```

👉 In interviews, PM2 is often mentioned as the **go-to production tool** for Node.js scaling.

---

5. Common Scaling Strategies

☑ Use **Cluster module** for multi-core utilization.
☑ Use **PM2** for process management & zero-downtime reloads.
☑ Use **NGINX/Load balancers** for horizontal scaling across servers.
☑ Use **Redis/Memcached** for distributed caching.
☑ Use **Docker & Kubernetes** for containerized microservices scaling.

---

6. Interview Questions

◇ Q1: How does Node.js handle multiple requests with a single thread?
👉 Event loop + async I/O. For CPU-bound tasks, use **worker threads / clustering**.

◇ Q2: What's the difference between Cluster and Worker Threads?
👉 **Cluster** = multiple Node processes, **Worker Threads** = multiple threads inside one process.

◇ Q3: How do you scale Node.js in production?
👉 Vertical scaling, clustering with PM2, horizontal scaling with load balancers, containerization (Docker/K8s).

☑ **Key Takeaway:**

- Node.js by default uses **one CPU core** → not fully efficient.
- **Cluster module** & **PM2** allow **multi-core utilization**.
- **Horizontal scaling with load balancers** ensures **high availability & resilience**.

Would you like me to cover **14. Worker Threads & Child Processes** next (goes deeper into handling CPU-intensive tasks in Node.js)?