Day 6

Case Study 1 — Node.js Clustering Using Cluster Module

Description

- Scenario: A Node.js API performs **CPU-intensive tasks** (e.g., calculating factorial).
- Single-threaded Node.js cannot handle multiple requests efficiently for CPU-heavy operations.
- Use **Node.js cluster module** to utilize **all CPU cores** for better concurrency.

Project Structure

```
cluster-app/
├── app.js
├── package.json
```

Step 1 — App Setup Using Cluster

app.js

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

const numCPUs = os.cpus().length;

if (cluster.isMaster) {
  console.log(`Master ${process.pid} is running`);

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

  cluster.on("exit", (worker, code, signal) => {
```

```
    console.log(`Worker ${worker.process.pid} died,
restarting...`);
    cluster.fork();
  });
} else {
  // Worker processes
  const server = http.createServer((req, res) => {
    if (req.url === "/factorial") {
      let n = 25;
      let result = 1;
      for (let i = 2; i <= n; i++) result *= i; // CPU-
intensive
      res.writeHead(200, { "Content-Type": "text/plain" });
      res.end(`Factorial of ${n} is ${result}`);
    } else {
      res.writeHead(200, { "Content-Type": "text/plain" });
      res.end("Hello from worker " + process.pid);
    }
  });

  server.listen(3000, () => console.log(`Worker $
{process.pid} started`));
}
```

Explanation

1. `cluster.isMaster` → master process forks workers equal to CPU cores.
2. Each worker handles incoming requests → **parallel processing**.
3. High concurrency is supported since requests are distributed among workers.
4. Master restarts workers if they crash.

Sample Output

```
Master 12345 is running
Worker 12346 started
Worker 12347 started
Worker 12348 started
Worker 12349 started
```

Request Output (GET /factorial):

```
Factorial of 25 is 15511210043330985984000000
```

- Multiple requests can be handled simultaneously by different workers.

Case Study 2 — Load Balancing with PM2

Description

- Scenario: A Node.js API handles **high volume requests** for product listing.
- Use **PM2 process manager** for clustering and built-in load balancing.
- Provides **monitoring, automatic restart, and log management**.

Project Structure

```
pm2-loadbalancer-app/
├── app.js
├── package.json
```

Step 1 — Install Dependencies

```
npm install express
npm install pm2 -g
```

Step 2 — App Setup

app.js

```
const express = require("express");
const app = express();

app.get("/products", (req, res) => {
  // Simulate delay
  const products = [
    { id: 1, name: "Laptop" },
    { id: 2, name: "Phone" },
    { id: 3, name: "Tablet" }
  ];
```

```
  setTimeout(() => res.json(products), 1000); // simulate I/O
delay
});

app.listen(3000, () => console.log(`Server running on port
3000`));
```

Step 3 — Start App with PM2

```
pm2 start app.js -i max --name "product-api"
```

- `-i max` → spawn one process per CPU core.
- PM2 load balances incoming requests automatically.

Step 4 — Monitor PM2

```
pm2 list
pm2 monit
pm2 logs product-api
```

Explanation

1. PM2 manages **multiple instances** of the app across CPU cores.
2. Incoming requests are **load balanced** across instances.
3. PM2 restarts crashed processes automatically → **high availability**.
4. Ideal for I/O-heavy APIs handling high concurrency.

Sample Output

PM2 List:

id	name	mode	pid	status	cpu	memory
0	product-api	fork	12346	online	1%	50 MB
1	product-api	fork	12347	online	1%	52 MB

GET /products Response:

```
[
  { "id": 1, "name": "Laptop" },
  { "id": 2, "name": "Phone" },
  { "id": 3, "name": "Tablet" }
]
```

- Multiple requests are distributed across processes → faster response under high load.

✅ **Key Concepts Covered**

1. **Case Study 1:** Node.js `cluster` module for CPU-bound apps → parallel processing using multiple cores.
2. **Case Study 2:** PM2 for load balancing, auto-restart, and monitoring → ideal for high-concurrency scenarios.

Case Study 1 — Node.js Clustering Using Cluster Module

Description

- Scenario: A Node.js API performs **CPU-intensive tasks** (e.g., calculating factorial).
- Single-threaded Node.js cannot handle multiple requests efficiently for CPU-heavy operations.
- Use **Node.js cluster module** to utilize **all CPU cores** for better concurrency.

Project Structure

```
cluster-app/
├── app.js
├── package.json
```

Step 1 — App Setup Using Cluster

app.js

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

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

const numCPUs = os.cpus().length;

if (cluster.isMaster) {
  console.log(`Master ${process.pid} is running`);

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

  cluster.on("exit", (worker, code, signal) => {
    console.log(`Worker ${worker.process.pid} died,
restarting...`);
    cluster.fork();
  });
} else {
  // Worker processes
  const server = http.createServer((req, res) => {
    if (req.url === "/factorial") {
      let n = 25;
      let result = 1;
      for (let i = 2; i <= n; i++) result *= i; // CPU-
intensive
      res.writeHead(200, { "Content-Type": "text/plain" });
      res.end(`Factorial of ${n} is ${result}`);
    } else {
      res.writeHead(200, { "Content-Type": "text/plain" });
      res.end("Hello from worker " + process.pid);
    }
  });

  server.listen(3000, () => console.log(`Worker $
{process.pid} started`));
}
```

Explanation

1. `cluster.isMaster` → master process forks workers equal to CPU cores.
2. Each worker handles incoming requests → **parallel processing**.
3. High concurrency is supported since requests are distributed among workers.
4. Master restarts workers if they crash.

Sample Output

```
Master 12345 is running
Worker 12346 started
Worker 12347 started
Worker 12348 started
Worker 12349 started
```

Request Output (GET /factorial):

```
Factorial of 25 is 15511210043330985984000000
```

- Multiple requests can be handled simultaneously by different workers.

Case Study 2 — Load Balancing with PM2

Description

- Scenario: A Node.js API handles **high volume requests** for product listing.
- Use **PM2 process manager** for clustering and built-in load balancing.
- Provides **monitoring, automatic restart, and log management**.

Project Structure

```
pm2-loadbalancer-app/
├── app.js
├── package.json
```

Step 1 — Install Dependencies

```
npm install express
npm install pm2 -g
```

Step 2 — App Setup

app.js

```
const express = require("express");
const app = express();

app.get("/products", (req, res) => {
  // Simulate delay
  const products = [
    { id: 1, name: "Laptop" },
    { id: 2, name: "Phone" },
    { id: 3, name: "Tablet" }
  ];
  setTimeout(() => res.json(products), 1000); // simulate I/O
delay
});

app.listen(3000, () => console.log(`Server running on port
3000`));
```

Step 3 — Start App with PM2

```
pm2 start app.js -i max --name "product-api"
```

- `-i max` → spawn one process per CPU core.
- PM2 load balances incoming requests automatically.

Step 4 — Monitor PM2

```
pm2 list
pm2 monit
pm2 logs product-api
```

Explanation

1. PM2 manages **multiple instances** of the app across CPU cores.
2. Incoming requests are **load balanced** across instances.
3. PM2 restarts crashed processes automatically → **high availability**.
4. Ideal for I/O-heavy APIs handling high concurrency.

Sample Output

PM2 List:

id	name	mode	pid	status	cpu	memory
0	product-api	fork	12346	online	1%	50 MB
1	product-api	fork	12347	online	1%	52 MB

GET /products Response:


```
[
  { "id": 1, "name": "Laptop" },
  { "id": 2, "name": "Phone" },
  { "id": 3, "name": "Tablet" }
]
```


- Multiple requests are distributed across processes → faster response under high load.

✅ Key Concepts Covered

1. **Case Study 1:** Node.js `cluster` module for CPU-bound apps → parallel processing using multiple cores.
2. **Case Study 2:** PM2 for load balancing, auto-restart, and monitoring → ideal for high-concurrency scenarios.