Day 5

Example 1 — Profiling CPU-Intensive Function

Scenario

- A Node.js app calculates **Fibonacci numbers recursively**, which is CPU-intensive.
- We want to **profile the function** to identify performance bottlenecks.

Project Structure

```
cpu-profile-app/
│── app.js
│── utils/
│    └── fibonacci.js
│── package.json
```

Step 1 — Fibonacci Function

utils/fibonacci.js

```
function fibonacci(n) {
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}

module.exports = { fibonacci };
```

Step 2 — App Setup

app.js

```
const { fibonacci } = require("./utils/fibonacci");

console.time("fib-35");
const result = fibonacci(35); // CPU-intensive
```

```
console.timeEnd("fib-35");

console.log("Fibonacci(35):", result);
```

Step 3 — Profiling with Node.js

Run with CPU profiler:

```
node --prof app.js
```

- Node generates a `isolate-0x*.log` file.
- Process with `node --prof-process isolate-0x*.log > processed.txt` to analyze hot functions.

Explanation

- `console.time` shows execution time for quick benchmarking.
- `--prof` provides detailed **CPU profiling** to identify which function consumes most CPU.

Sample Output

```
fib-35: 1200ms
Fibonacci(35): 9227465
```

- `processed.txt` shows `fibonacci` function as the hot spot → candidate for optimization (e.g., memoization).

Example 2 — Benchmarking Async I/O Performance

Scenario

- Node.js app reads **large files asynchronously** using **fs.readFile** vs **streams**.
- Benchmark both approaches to see which is faster and uses less memory.

Project Structure

```
io-benchmark-app/
│── app.js
│── package.json
│── largeFile.txt
```

Step 1 — Install Dependencies

```
npm install benchmark
```

Step 2 — App Setup

app.js

```
const fs = require("fs");
const Benchmark = require("benchmark");

const filePath = "./largeFile.txt";

const suite = new Benchmark.Suite();

// Async readFile
suite.add("fs.readFile", {
  defer: true,
  fn: deferred => {
    fs.readFile(filePath, "utf8", (err, data) => {
      if (err) throw err;
      deferred.resolve();
    });
  }
});

// Stream read
suite.add("fs.createReadStream", {
  defer: true,
  fn: deferred => {
    const stream = fs.createReadStream(filePath, "utf8");
    stream.on("data", () => {});
    stream.on("end", () => deferred.resolve());
  }
});

// Run benchmarks
```

```
suite
  .on("cycle", event => console.log(String(event.target)))
  .on("complete", function () {
    console.log("Fastest is " +
this.filter("fastest").map("name"));
  })
  .run({ async: true });
```

Explanation

1. `Benchmark.Suite()` measures **time taken for each approach**.
2. `fs.readFile` → loads entire file into memory.
3. `fs.createReadStream` → reads file in chunks → less memory usage.
4. `.defer = true` → handles asynchronous operations correctly.

Sample Output

```
fs.readFile x 5.12 ops/sec ±0.85% (10 runs sampled)
fs.createReadStream x 20.47 ops/sec ±1.12% (15 runs sampled)
Fastest is fs.createReadStream
```

- Shows **streaming is faster** and more memory-efficient for large files.

✅ Key Concepts Covered

- **Example 1:** CPU profiling for heavy computation functions.
- **Example 2:** Benchmarking I/O operations to identify performance bottlenecks.

Example 1 — Redis Caching for API Responses

Scenario

- A Node.js API fetches **employee data** from a database (simulated).
- To reduce DB load, we cache the **GET /employees** response in **Redis**.

Project Structure

```
redis-cache-api/
│── app.js
│── routes/
│     └── employees.js
│── package.json
```

Step 1 — Install Dependencies

```
npm install express redis
```

Step 2 — Employee Routes with Redis Cache

routes/employees.js

```
const express = require("express");
const router = express.Router();
const redis = require("redis");

const client = redis.createClient();
client.connect();

const employees = [
  { id: 1, name: "Alice", department: "HR" },
  { id: 2, name: "Bob", department: "IT" }
];

router.get("/", async (req, res) => {
  try {
    // Check Redis cache
    const cached = await client.get("employees");
    if (cached) return res.json(JSON.parse(cached));

    // Simulate DB fetch
    const data = employees;

    // Store in Redis for 60 seconds
    await client.setEx("employees", 60,
JSON.stringify(data));

    res.json(data);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});

module.exports = router;
```

Step 3 — App Setup

app.js

```
const express = require("express");
const employeeRoutes = require("./routes/employees");

const app = express();
app.use("/employees", employeeRoutes);

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

Explanation

1. `client.get("employees")` → checks if data exists in Redis.
2. `client.setEx("employees", 60, ...)` → caches data for **60 seconds**.
3. Reduces repeated DB calls → improves performance.

Sample Output

1st request (cache miss):

```
[
  { "id": 1, "name": "Alice", "department": "HR" },
  { "id": 2, "name": "Bob", "department": "IT" }
]
```

2nd request (cache hit within 60s):

```
[
  { "id": 1, "name": "Alice", "department": "HR" },
  { "id": 2, "name": "Bob", "department": "IT" }
]
```

- Notice response is returned faster for cached requests.

Example 2 — HTTP Cache Headers + In-Memory Cache

Scenario

- A **news API** returns articles.
- Use **in-memory cache** and **HTTP cache headers** to optimize client-side caching.

Project Structure

```
http-cache-api/
│── app.js
│── routes/
│    └── articles.js
│── package.json
```

Step 1 — Install Dependencies

```
npm install express
```

Step 2 — Articles Route with In-Memory + HTTP Cache

routes/articles.js

```
const express = require("express");
const router = express.Router();

let cache = null;
let cacheTime = null;

const articles = [
  { id: 1, title: "Node.js Tips" },
  { id: 2, title: "Express.js Middleware" }
];
```

```
router.get("/", (req, res) => {
  const now = Date.now();

  // Serve from in-memory cache if not expired
  if (cache && now - cacheTime < 30 * 1000) {
    res.set("Cache-Control", "public, max-age=30"); // HTTP
cache header
    return res.json(cache);
  }

  // Simulate DB fetch
  cache = articles;
  cacheTime = now;

  res.set("Cache-Control", "public, max-age=30");
  res.json(articles);
});

module.exports = router;
```

Step 3 — App Setup

app.js

```
const express = require("express");
const articleRoutes = require("./routes/articles");

const app = express();
app.use("/articles", articleRoutes);

app.listen(3001, () => console.log("Server running on port
3001"));
```

Explanation

1. `cache` stores data in-memory to avoid repeated DB calls.
2. `Cache-Control: public, max-age=30` → instructs clients/browser to cache response for **30 seconds**.
3. Reduces network latency and server load.

Sample Output

1st request (cache miss):

```
[
  { "id": 1, "title": "Node.js Tips" },
  { "id": 2, "title": "Express.js Middleware" }
]
```

2nd request within 30 seconds (cache hit):

- Data served from in-memory cache.
- HTTP response header:

```
Cache-Control: public, max-age=30
```

✅ **Key Concepts Covered**

- **Example 1:** Redis caching for server-side performance optimization.
- **Example 2:** In-memory + HTTP cache headers for client-side caching.
- Both reduce repeated database calls and improve response times.

Example 1 — File Upload Using Streams

Scenario

- We want to **upload large files** without loading them entirely into memory.
- Use **streams** to save the file directly to disk.

Project Structure

```
stream-upload-api/
│── app.js
│── uploads/          # directory to store uploaded files
│── package.json
```

Step 1 — Install Dependencies

```
npm install express multer
```

Step 2 — App Setup with Stream Upload

app.js

```
const express = require("express");
const multer = require("multer");
const fs = require("fs");
const path = require("path");

const app = express();
const upload = multer({ dest: "uploads/" }); // temp storage

// POST /upload
app.post("/upload", upload.single("file"), (req, res) => {
  const tempPath = req.file.path;
  const targetPath = path.join(__dirname, "uploads",
req.file.originalname);

  // Stream file from temp location to final destination
  const readStream = fs.createReadStream(tempPath);
  const writeStream = fs.createWriteStream(targetPath);

  readStream.pipe(writeStream);

  writeStream.on("finish", () => {
    // Delete temp file
    fs.unlinkSync(tempPath);
    res.json({ message: "File uploaded successfully",
filename: req.file.originalname });
  });

  writeStream.on("error", (err) => {
    res.status(500).json({ error: err.message });
  });
});

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

Explanation

1. **Multer** handles file upload to a temporary folder.

2. **fs.createReadStream + fs.createWriteStream** → moves file efficiently using streams.
3. Streams prevent **high memory usage** for large files.

Sample Output

POST /upload (file: `large-file.txt`)

```
{ "message": "File uploaded successfully", "filename":
"large-file.txt" }
```

- File saved in `uploads/large-file.txt`.

Example 2 — File Download Using Streams

Scenario

- Provide an API endpoint to **download large files** efficiently.
- Use **read streams** to avoid loading the entire file into memory.

Project Structure

```
stream-download-api/
│── app.js
│── files/          # store files for download
│── package.json
```

Step 1 — App Setup with Stream Download

app.js

```
const express = require("express");
const fs = require("fs");
const path = require("path");

const app = express();

// GET /download/:filename
app.get("/download/:filename", (req, res) => {
  const filePath = path.join(__dirname, "files",
req.params.filename);

  if (!fs.existsSync(filePath)) return res.status(404).json({
error: "File not found" });

  // Stream file to client
  const readStream = fs.createReadStream(filePath);

  res.setHeader("Content-Disposition", `attachment;
filename="${req.params.filename}"`);
  readStream.pipe(res);

  readStream.on("error", (err) => {
    res.status(500).json({ error: err.message });
  });
});

app.listen(3001, () => console.log("Server running on port
3001"));
```

Explanation

1. **fs.createReadStream** streams file to response object.
2. **res.setHeader("Content-Disposition")** → triggers download in browser.
3. Efficient for **large files** since memory usage is low.

Sample Output

GET /download/sample.pdf → browser downloads `sample.pdf` directly.

- No large memory usage even for multi-GB files.

✅ Key Concepts Covered

1. **Example 1 (Upload)**: Use **streams to handle file uploads** efficiently.
2. **Example 2 (Download)**: Use **streams to serve files** efficiently without loading the entire file into memory.
3. Streams are critical for **performance optimization** in Node.js apps dealing with large data.