MicroHealth

Microservice Health Monitoring System

Developed by Pranay

Project Documentation & Technical Report

January 2026

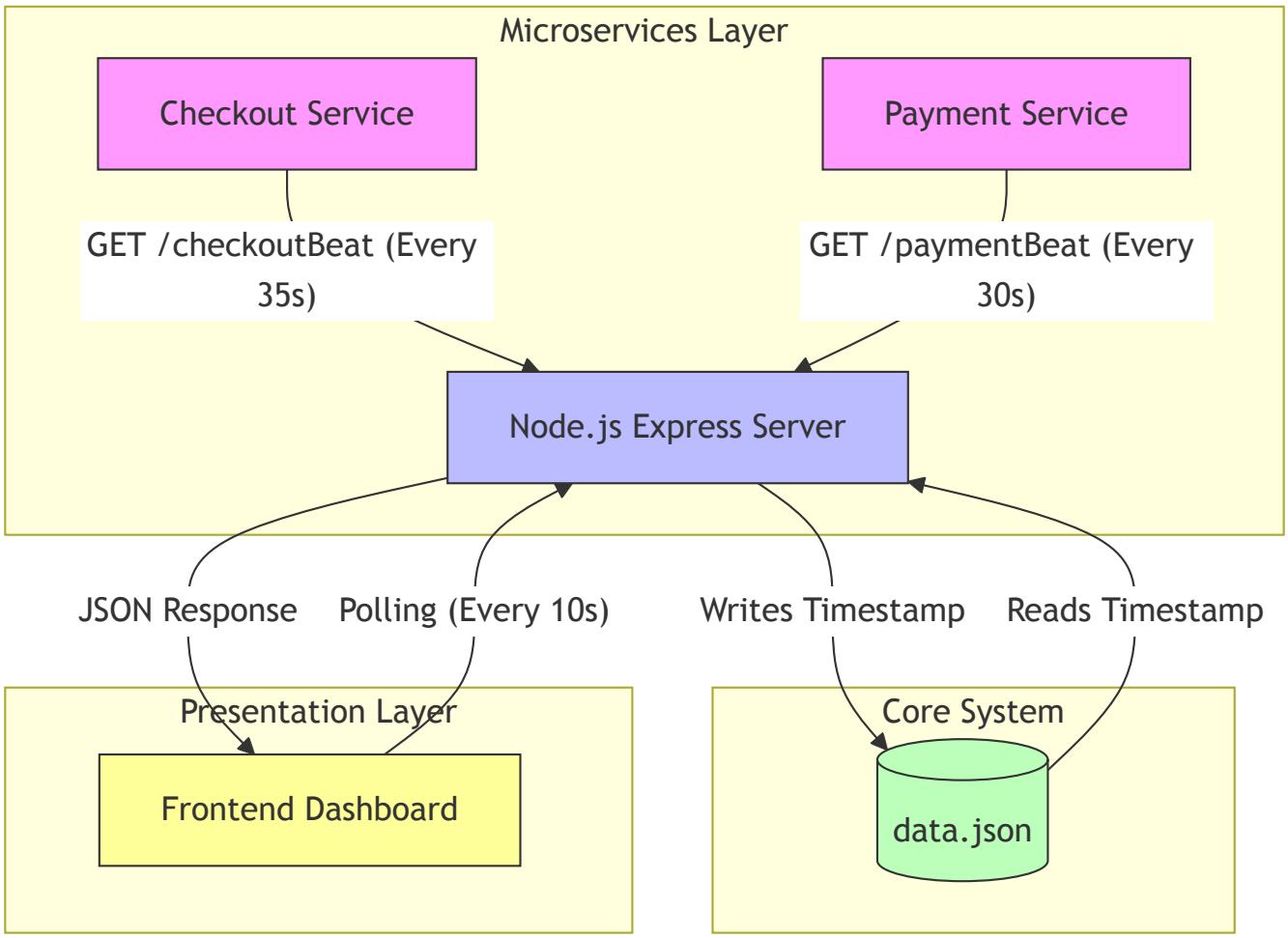
1. Executive Summary

MicroHealth is a robust architectural prototype demonstrating the **"Heartbeat" pattern** in distributed systems. It solves the critical problem of *Service Discovery and Health Monitoring* in a microservices environment.

In a real-world scenario with hundreds of services (Payment, Auth, Checkout, etc.), it is impossible to manually check if they are running. MicroHealth implements a centralized **Health Check Server** that listens for "heartbeats" (regular HTTP signals) from satellite services. If a service stops sending heartbeats (due to crash or network failure), the system automatically flags it as **DOWN**.

2. System Architecture

High-Level Data Flow



Technical Stack Logic

- **Node.js & Express**: Chosen for non-blocking I/O, perfect for handling high-frequency heartbeat requests without lag.
- **JSON Filesystem DB**: Used here for simplicity and zero-configuration persistence. In production, this would be Redis or a Time-Series Database.
- **Axios**: Robust HTTP client used over `fetch` for its automatic JSON parsing and better error handling.
- **Vanilla JS Frontend**: Lightweight dashboard without the overhead of React/Angular, ensuring instant loading.

3. Component Deep Dive

A. The Central Server (`index.js`)

The "Brain" of the system. It exposes two types of endpoints:

1. **Writer Endpoint** (`/:serviceName`): Captures the heartbeat.
2. **Reader Endpoint** (`/get/:serviceName`): Serves the status.

> **Key Code Pattern:** Dynamic Routing using `req.params` allows us to add infinite new services (Delivery, Auth, Email) without changing the server code.

```
 // Dynamic Route Handler
app.get("/:serviceName",(req,res)=>{
    // 1. Capture the service name from URL
    let {serviceName} = req.params;

    // 2. Update In-Memory Object
    data[serviceName] = Date.now();

    // 3. Persist to Disk (Critical for Crash Recovery)
    fs.writeFileSync("./data.json", JSON.stringify(data,null,2));

    res.send("YES");
});
```

B. The Satellite Services (`MICROSERVICES/`)

These represent the actual business logic containers. They run autonomously.

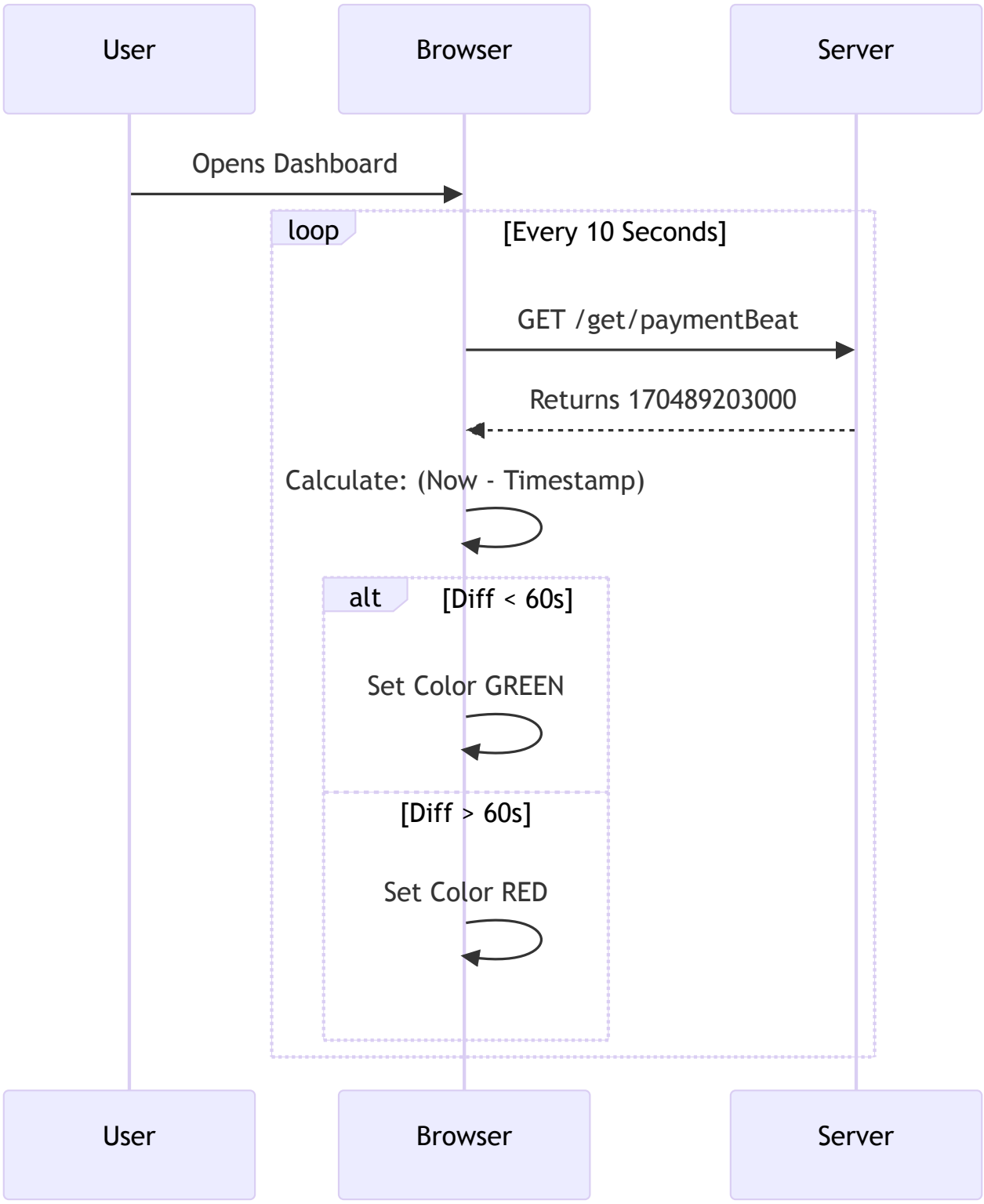
The "Heartbeating" Mechanism:

They use `setInterval` to create an infinite loop. This is similar to a *Daemon Process*. Notice `checkout.js` sends every 35s and `payment.js` every 30s. This variation tests the server's

ability to handle asynchronous updates.

C. The Dashboard (`public/app.js`)

The "Eyes" of the user. It uses the **Polling Technique**.



4. Advanced Technical Interview Questions

Be prepared to answer these deep-dive questions using this project as a reference.

Q1: Why use `fs.writeFileSync` instead of `writeFile`?

Answer: In this specific prototype, we use `Sync` to ensure data consistency. We don't want to read the file before the previous write is finished (Race Condition). However, in a high-scale production app, blocking the Event Loop with Sync operations is bad practice. We would replace this with a Database (MongoDB/Redis) or use proper async/await with file locking.

Q2: What is "Polling" vs "WebSockets"? Why did you choose Polling?

Answer:

- **Polling (Used here)**: The client asks the server "Any news?" every X seconds. Easiest to implement. Good for "Status" checks where millisecond latency doesn't matter.
- **WebSockets**: The server pushes data to the client instantly. Better for Chat apps or Stock tickers.

I chose Polling because keeping a persistent WebSocket connection for a health dashboard is unnecessary overhead. Updates every 10 seconds are acceptable.

Q3: How would you scale this?

Answer: 1. **Database**: Move from `data.json` to Redis (In-memory store) for extremely fast reads/writes.
2. **Load Balancer**: If we had 10,000 services, one Node server implies a bottleneck. We'd put Nginx in front of multiple Node instances.
3. **Alerting**: Instead of just turning Red, the system should integrate Twilio or PagerDuty APIs to SMS the DevOps team when a service dies.

5. Setup & Usage Guide

1. **Prerequisites**: Install Node.js.

2. **Installation**: Run `npm install` to get Express and Axios.
3. **Start Server**:

   ```
   node index.js
   ```

   Server runs on `http://localhost:3000`.
4. **Start Services**: Open new terminals for each:

   ```
   node MICROSERVICES/payment.js
   node MICROSERVICES/checkout.js
   ```

5. **Monitor**: Open `http://localhost:3000` in your browser.