

Sri Lanka Institute of Information Technology



Visual Analytics and User Experience Design (IT4031)

Assignment 2

Final Report

Group ID: 2025_A2_G27

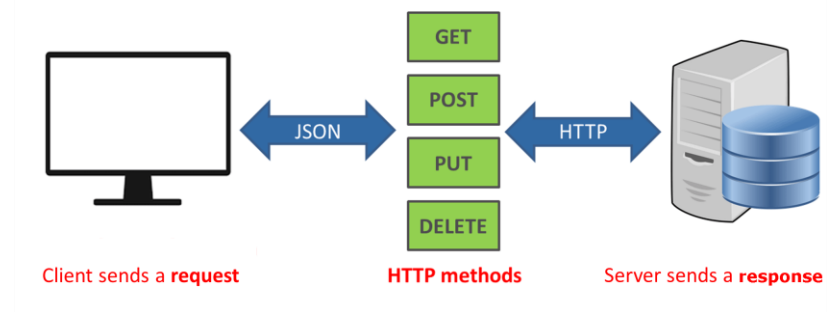
	Student Registration Number	Student Name
1	IT21468360	Rizvi F. A
2	IT21489464	Navojith T
3	IT21924750	Senevirathna W.P. U
4	IT21469046	Adhikari A.M.N.H.

1. Introduction

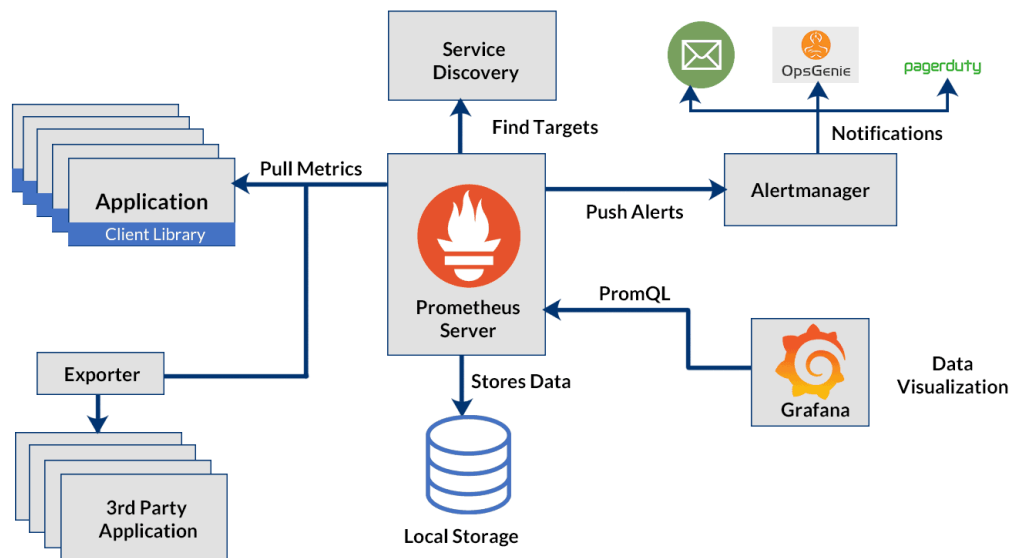
This report presents the development and deployment of a full-stack observability solution for an Employee Management REST API using Prometheus and Grafana. The system was designed to demonstrate real-world principles of modern application monitoring, API design, metric exposure, and alert-driven observability using open-source tools.

The project incorporates several foundational concepts:

- **RESTful API:** A standardized web service interface that supports Create, Read, Update, and Delete (CRUD) operations. This enables structured communication between client and server.

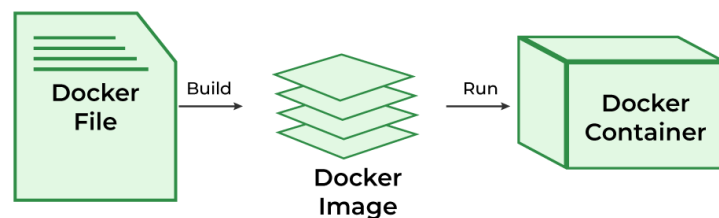


- **Prometheus:** A time-series monitoring tool used to scrape and store metrics exposed by the API. It allows real-time insights into system behavior and supports queries using PromQL.



- **Metrics:** Quantifiable data points (like request count, error rate, and request duration) used to assess system health and performance. These are collected both automatically and through custom instrumentation.

- **Grafana:** A powerful visualization platform that integrates with Prometheus to display metrics on dashboards. It supports diverse chart types and enables users to track trends visually.
- **Alerting:** The ability to define rules that automatically trigger warnings or alerts based on metric thresholds. Grafana's unified alerting system handles rule evaluation and provides visibility into critical events.
- **Docker Compose:** A container orchestration tool that allows all services (API, Prometheus, Grafana, Node Exporter) to run together in isolated but coordinated containers, simplifying setup and deployment.



By integrating these concepts, the project provides a complete observability solution suitable for small-scale enterprise applications and educational demonstrations.

What is Observability?

Observability is the ability to understand what is happening inside a system based on the data it exposes, even if you cannot interact with it directly.

In simple terms, it means:

“Can I tell what’s going wrong (or right) in my application just by looking at the data it outputs?”

Why Do We Implement Observability?

Modern applications run in complex, distributed environments. Without observability:

- You don’t know if your app is working correctly
- You can't identify slowdowns, errors, or abnormal usage
- You can’t detect issues before users complain

Benefits of Adding Observability

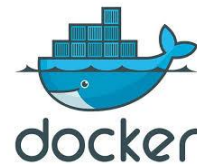
Aspect	Benefit
Metrics	Track how often employees are added, updated, deleted
Request tracking	Monitor how many users call the API, and which endpoints
Latency tracking	Identify slow requests or overloaded endpoints
Alerts	Get notified when error counts or traffic spikes occur
Visualization	Use Grafana to see everything in real-time
Debugging	Use Prometheus to troubleshoot after an issue has happened

2. Objective

The primary objective of this project was to develop a RESTful Employee Management API and integrate it with a full observability stack using Prometheus and Grafana. The goal was not only to implement CRUD operations but also to expose and monitor key application metrics, visualize performance data, and automate alerting critical events. The system was deployed using Docker Compose to ensure ease of setup, reproducibility, and service orchestration across different environments.

3. Technology Stack

- **Node.js + Express:** Used to build the core RESTful API that handles CRUD operations for employee records.
- **Prometheus:** Acts as the monitoring backend by scraping exposed metrics from the API and collecting time-series data.
- **Grafana:** Visualizes metrics from Prometheus using customizable dashboards and provides alerting capabilities based on thresholds.
- **Docker & Docker Compose:** Enables containerization and orchestrated deployment of all services (API, Prometheus, Grafana, Node Exporter).
- **Node Exporter:** An additional service that collects system-level metrics (CPU, memory, disk) for Grafana to visualize.

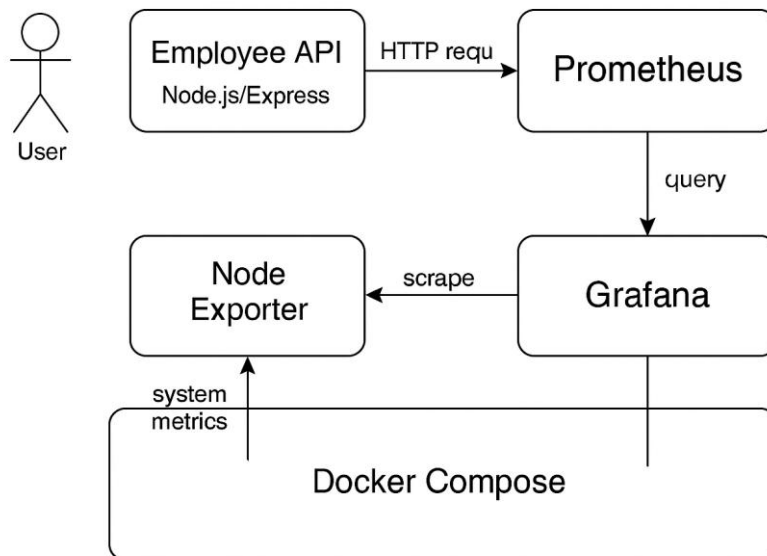


4. System Architecture

The observability system is composed of the following components, orchestrated using Docker Compose:

- **Employee API:** A Node.js/Express service that handles HTTP requests related to employee data. It also exposes Prometheus-compatible metrics via the /metrics endpoint.
- **Prometheus:** Scrapes metrics from the API and Node Exporter at regular intervals and store them as time-series data.
- **Grafana:** Connects to Prometheus as a data source and displays the metrics in rich dashboards. It also evaluates alert rules based on defined thresholds.
- **Node Exporter:** Provides operating system metrics like CPU usage, memory, and disk I/O.
- **Docker Compose:** Manages the lifecycle and network configuration of all services, enabling consistent and isolated environments.

The diagram above illustrates how the services interact. Prometheus pulls data from both the API and Node Exporter, and Grafana queries Prometheus to power the dashboards and alerts. All components are deployed and managed together using Docker Compose.



5. API Design and Implementation

5.1. Building the API – server.js

The backend of the system was developed using Node.js with the Express.js framework, forming a RESTful API that supports full CRUD operations. This API allows clients to create, read, update, and delete employee records, making it functionally complete for use cases like HR systems or staff databases.

Core API Endpoints Implemented:

1. **GET /employees:** Returns a list of all employees in JSON format.
2. **GET /employees/:id:** Returns details of a specific employee by ID.
3. **POST /employees:** Accepts new employee details (name, position, salary) and creates a new record.
4. **PUT /employees/:id:** Updates fields (name, position, salary) of an existing employee.
5. **DELETE /employees/:id:** Deletes an employee from the dataset.

Prometheus Metric Integration

To add observability to the system, the prom-client library was integrated. This allowed the API to track and expose internal behavior through Prometheus-compatible metrics.

Metrics Implemented:

- **http_requests_total:** Counts every API request with method and endpoint labels.
- **employees_added_total:** Increments every time a new employee is added.
- **employees_updated_total:** Tracks employee updates.
- **employees_deleted_total:** Tracks deletions.
- **errors_total:** Increments on invalid actions like missing fields or non-existent IDs.
- **http_request_duration_seconds:** Captures response latency in histogram buckets (e.g., 0.1s, 0.5s, 1s).

Data Persistence (Why data.json Matters)

Instead of losing employee data after every container restart, the system writes all employees and metric counters into a data.json file. At startup, if this file exists, the API loads its contents, thus restoring previous data and continuing metric tracking without reset.

Metrics Endpoint for Prometheus

The API exposes a special endpoint: **GET /metrics**

This serves all Prometheus metrics in plaintext format. Prometheus scrapes this endpoint at regular intervals (every 5 seconds in your setup).

```
// Prometheus Metric Configuration

client.collectDefaultMetrics();

const requestCounter = new client.Counter({
  name: 'http_requests_total',
  help: 'Total number of HTTP requests',
  labelNames: ['method', 'endpoint']
});

const employeeAddedCounter = new client.Counter({
  name: 'employees_added_total',
  help: 'Total number of employees added'
});

const employeeUpdatedCounter = new client.Counter({
  name: 'employees_updated_total',
  help: 'Total number of employee updates'
});

const employeeDeletedCounter = new client.Counter({
  name: 'employees_deleted_total',
  help: 'Total number of employee deletions'
});

const errorCounter = new client.Counter({
  name: 'errors_total',
  help: 'Total number of errors (e.g., not found, bad request)'
});

const httpRequestDurationSeconds = new client.Histogram({
  name: 'http_request_duration_seconds',
  help: 'Duration of HTTP requests in seconds',
  labelNames: ['method', 'route', 'code'],
  buckets: [0.1, 0.3, 0.5, 1, 1.5, 2, 5] // seconds
});

// Restore counter values
employeeAddedCounter.inc(counterState.employees_added_total);
employeeUpdatedCounter.inc(counterState.employees_updated_total);
employeeDeletedCounter.inc(counterState.employees_deleted_total);
errorCounter.inc(counterState.errors_total);
```

Figure 1: API Initialization and Data Persistence

Figure 1 displays the top portion of the `server.js` file, where the Express application is initialized. Essential modules such as `express`, `prom-client`, `fs`, and `path` are imported. The app is configured to parse JSON request bodies and define the path to the `data.json` file, which serves as persistent storage. If the file exists, its contents are loaded into the application to recover previously saved employee data and Prometheus counter values. This mechanism ensures that the system resumes from the last known state even after a container restarts.

Figure 2 shows the configuration of Prometheus metrics using the `prom-client` module. Several custom counters are defined: `http_requests_total` (tracks all HTTP requests), `employees_added_total`, `employees_updated_total`, `employees_deleted_total`, and `errors_total`. Additionally, a histogram named `http_request_duration_seconds` is defined to monitor API response time in various time buckets. At startup, these counters are incremented based on saved values from `data.json` to restore historical state. This metric setup allows Prometheus to monitor both the functional behavior and performance characteristics of the API.

```
// Middleware to Track Requests Automatically

app.use((req, res, next) => {
  const end = httpRequestDurationSeconds.startTimer();
  res.on('finish', () => {
    const routePath = req.route?.path || req.path;
    requestCounter.labels(req.method, routePath).inc();
    end({ method: req.method, route: routePath, code: res.statusCode });
  });
  next();
});

// API ROUTES

app.get('/employees', (req, res) => {
  res.json(employees);
});

app.get('/employees/:id', (req, res) => {
  const employee = employees[req.params.id];
  if (!employee) {
    errorCounter.inc();
    counterState.errors_total++;
    saveData();
    return res.status(404).json({ error: 'Employee not found' });
  }
  res.json(employee);
});

app.post('/employees', (req, res) => {
  const { name, position, salary } = req.body;
  if (!name || !position || !salary) {
    errorCounter.inc();
    counterState.errors_total++;
    saveData();
    return res.status(400).json({ error: 'Name, position, and salary are required.' });
  }
  const id = nextId++;
  employees[id] = { name, position, salary };
  employeeAddedCounter.inc();
  counterState.employees_added_total++;
  saveData();
  res.status(201).json({ id, ...employees[id] });
});
```

Figure 2: Prometheus Metric Configuration

```

app > data.json > ...
1  {
2    "employees": {
3      "1": {
4        "name": "John Doe",
5        "position": "Manager",
6        "salary": 65000
7      },
8      "2": {
9        "name": "Jane Smith",
10       "position": "Developer",
11       "salary": 55000
12     },
13     "3": {
14       "name": "Alice",
15       "position": "HR",
16       "salary": 65000
17     },
18     "6": {
19       "name": "Ayush",
20       "position": "Data Analyst",
21       "salary": 85000
22     },
23     "7": {
24       "name": "Mithuni",
25       "position": "Data Scientist",
26       "salary": 90000
27     },
28     "8": {
29       "name": "Rada",
30       "position": "HR",
31       "salary": 50000
32     }
33   },
34   "nextId": 11,
35   "counterState": {
36     "employees_added_total": 8,
37     "employees_updated_total": 4,
38     "employees_deleted_total": 2,
39     "errors_total": 13
40   }
41 }

```

Figure 3: Data.json File

```

app.put('/employees/:id', (req, res) => {
  const employee = employees[req.params.id];
  if (!employee) {
    errorCounter.inc();
    counterState.errors_total++;
    saveData();
    return res.status(404).json({ error: 'Employee not found' });
  }
  const { name, position, salary } = req.body;
  if (name) employee.name = name;
  if (position) employee.position = position;
  if (salary) employee.salary = salary;
  employeeUpdatedCounter.inc();
  counterState.employees_updated_total++;
  saveData();
  res.json(employee);
});

app.delete('/employees/:id', (req, res) => {
  if (!employees[req.params.id]) {
    errorCounter.inc();
    counterState.errors_total++;
    saveData();
    return res.status(404).json({ error: 'Employee not found' });
  }
  delete employees[req.params.id];
  employeeDeletedCounter.inc();
  counterState.employees_deleted_total++;
  saveData();
  res.status(204).send();
});

app.get('/metrics', async (req, res) => {
  res.set('Content-Type', client.register.contentType);
  res.end(await client.register.metrics());
});

function saveData() {
  fs.writeFileSync(DATA_FILE, JSON.stringify({ employees, nextId, counterState }, null, 2));
}

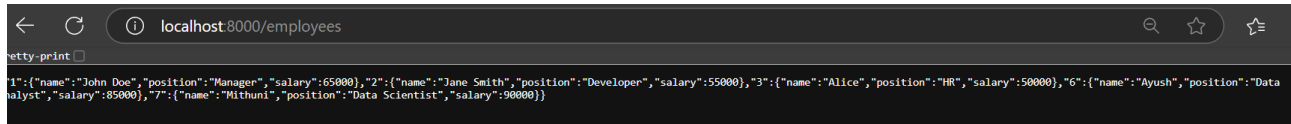
app.listen(port, () => {
  console.log('Employee Management API running at http://localhost:${port}');
});

```

Figure 4: Middleware and Core API Routes


The data.json file acts as the persistent storage mechanism for the Employee Management API. It contains two main sections: employees and counterState. The employees object holds all the employee records indexed by unique IDs. Each employee entry includes a name, position, and salary.

Figure 4 contains the logic responsible for request tracking and API functionality. Middleware is applied to every request to start a timer and count requests per route and method using the requestCounter. The GET /employees endpoint returns all employee records, while GET /employees/:id and POST /employees handle individual lookup and creation. If a request fails due to missing data or a non-existent employee ID, the errors_total counter is incremented, and a relevant error response is returned. This implementation ensures all interactions with the API are tracked and observable.



```
localhost:8000/employees
[{"name":"John Doe","position":"Manager","salary":65000}, {"name":"Jane Smith","position":"Developer","salary":55000}, {"name":"Alice","position":"HR","salary":50000}, {"name":"Ayush","position":"Data Analyst","salary":85000}, {"name":"Mithun","position":"Data Scientist","salary":90000}]
```

This confirms the successful execution of the GET /employees endpoint. It displays a JSON object containing multiple employee records, indicating that the in-memory data or the content loaded from data.json is being correctly served by the API. It also shows that the API is accessible via the browser and is functioning as expected.



```
localhost:8000/metrics
# HELP process_cpu_user_seconds_total Total user CPU time spent in seconds.
# TYPE process_cpu_user_seconds_total counter
process_cpu_user_seconds_total 4.834600000000002

# HELP process_cpu_system_seconds_total Total system CPU time spent in seconds.
# TYPE process_cpu_system_seconds_total counter
process_cpu_system_seconds_total 3.4939969999999985

# HELP process_cpu_seconds_total Total user and system CPU time spent in seconds.
# TYPE process_cpu_seconds_total counter
process_cpu_seconds_total 8.328596999999997

# HELP process_start_time_seconds Start time of the process since unix epoch in seconds.
# TYPE process_start_time_seconds gauge
process_start_time_seconds 1746706274

# HELP process_resident_memory_bytes Resident memory size in bytes.
# TYPE process_resident_memory_bytes gauge
process_resident_memory_bytes 61657088

# HELP process_virtual_memory_bytes Virtual memory size in bytes.
# TYPE process_virtual_memory_bytes gauge
process_virtual_memory_bytes 618971136

# HELP process_heap_bytes Process heap size in bytes.
# TYPE process_heap_bytes gauge
process_heap_bytes 65695744

# HELP process_open_fds Number of open file descriptors.
# TYPE process_open_fds gauge
process_open_fds 21

# HELP process_max_fds Maximum number of open file descriptors.
# TYPE process_max_fds gauge
process_max_fds 1048576

# HELP nodejs_eventloop_lag_seconds Lag of event loop in seconds.
# TYPE nodejs_eventloop_lag_seconds gauge
nodejs_eventloop_lag_seconds 0.024540367

# HELP nodejs_eventloop_lag_min_seconds The minimum recorded event loop delay.
# TYPE nodejs_eventloop_lag_min_seconds gauge
nodejs_eventloop_lag_min_seconds 0.009191424

# HELP nodejs_eventloop_lag_max_seconds The maximum recorded event loop delay.
# TYPE nodejs_eventloop_lag_max_seconds gauge
nodejs_eventloop_lag_max_seconds 0.012500991

# HELP nodejs_eventloop_lag_mean_seconds The mean of the recorded event loop delays.
# TYPE nodejs_eventloop_lag_mean_seconds gauge
nodejs_eventloop_lag_mean_seconds 0.010203489380392157
```

This image captures the content of the /metrics endpoint accessed through a web browser. It shows Prometheus-exposed metrics such as process_cpu_seconds_total, nodejs_eventloop_lag_seconds, and other application and system-level metrics. These metrics are displayed in the standard Prometheus exposition format, confirming that the API is acting as a valid Prometheus exporter and is ready for scraping and visualization in Prometheus and Grafana.

5.2. Setting Up Prometheus

To monitor application-specific and system-level metrics in real-time, Prometheus was configured as the metrics collector and time-series database. Prometheus acts as the central hub for gathering data from different exporters, enabling query execution, data retention, and alert evaluation.

The configuration for Prometheus was defined in a file located at `prometheus/prometheus.yml`. This YAML file specifies which endpoints Prometheus should scrape and how frequently. As shown, two scrape jobs were configured:

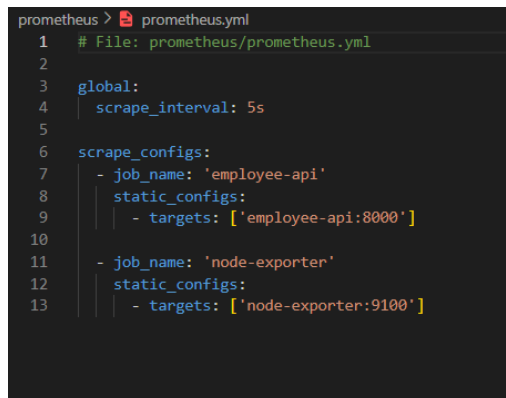
1. **employee-api**

- Scrapes metrics from: `http://employee-api:8000/metrics`
- This endpoint is served by the Node.js backend using the `prom-client` library and exposes counters such as `employees_added_total`, `errors_total`, etc.

2. **node-exporter**

- Scrapes system-level metrics from: `http://node-exporter:9100/metrics`
- This service provides information like CPU usage, memory, disk IO, and host uptime.

The `scrape_interval` was set to 5 seconds, which means Prometheus will query the `/metrics` endpoints every 5 seconds, ensuring near real-time updates of all metrics displayed in Grafana and available for alerting.

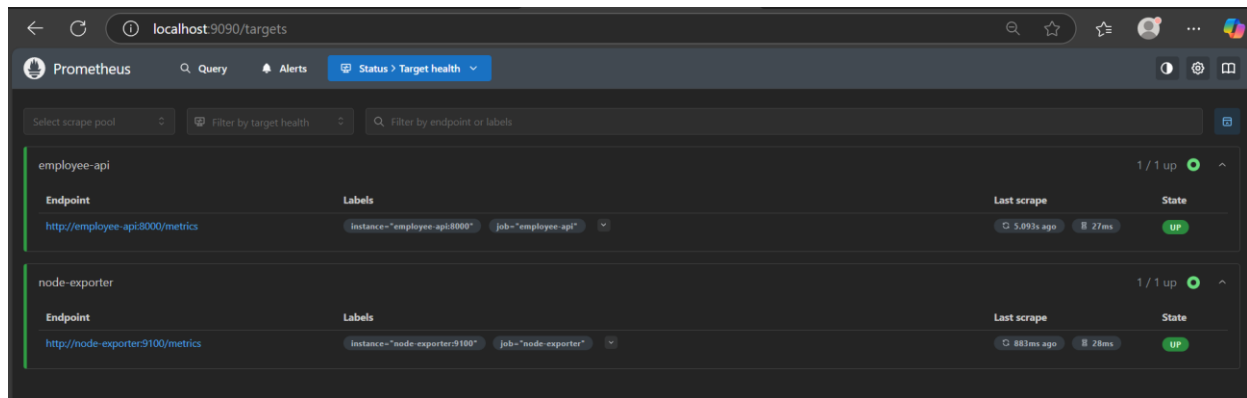
A screenshot of a code editor showing the `prometheus.yml` configuration file. The editor has a dark background with light-colored text. The file content is as follows:

```
1 # File: prometheus/prometheus.yml
2
3 global:
4   scrape_interval: 5s
5
6 scrape_configs:
7   - job_name: 'employee-api'
8     static_configs:
9       - targets: ['employee-api:8000']
10
11   - job_name: 'node-exporter'
12     static_configs:
13       - targets: ['node-exporter:9100']
```

This shows the actual `prometheus.yml` configuration file. The top-level `global` block defines the global scraping frequency. The `scrape_configs` section contains two jobs:

- The **employee-api job** targets the internal Docker hostname `employee-api:8000`.
- The **node-exporter job** targets `node-exporter:9100`.

These hostnames match the container names declared in `docker-compose.yml`, enabling service-to-service communication inside the Docker network.



This interface confirms whether Prometheus can successfully reach and scrape the configured endpoints.

As seen in the image:

- Both employee-api and node-exporter are marked as **UP**, meaning Prometheus is actively collecting data from them.
- The "Last scrape" field indicates the time since the last data collection.
- Response times (e.g., 27ms, 28ms) confirm that both endpoints are responding quickly and reliably.

This visual confirmation is essential to ensure the observability pipeline is functioning correctly.

PromQL Query Testing

Tested metric collection using Prometheus's built-in query language (PromQL). Example queries include:

- `rate(employees_added_total[1m])`: calculates the rate of employee additions per minute.
- `sum(http_requests_total) by (method)`: shows how many requests each HTTP method has handled.

These queries verified that metrics were being ingested correctly and helped prepare visualizations and alerts in Grafana.

Why Prometheus?

- It is an open-source, production-grade metrics collector.
- It supports multidimensional metrics through label-based filtering.
- It seamlessly integrates with Grafana for visualization.
- It supports alerting rules based on real-time metrics.

5.3. Creating Grafana Dashboards

Grafana was used as the primary frontend tool to visualize all metrics collected by Prometheus. The goal was to transform raw numerical metrics into intuitive, real-time visual representations that can guide operational awareness and performance monitoring.

Steps taken:

- Opened Grafana at <http://localhost:3000>
- Logged in as admin / vaude
- Added Prometheus as a data source
- Imported a custom dashboard (grafana/dashboard.json)

Visualizations created:

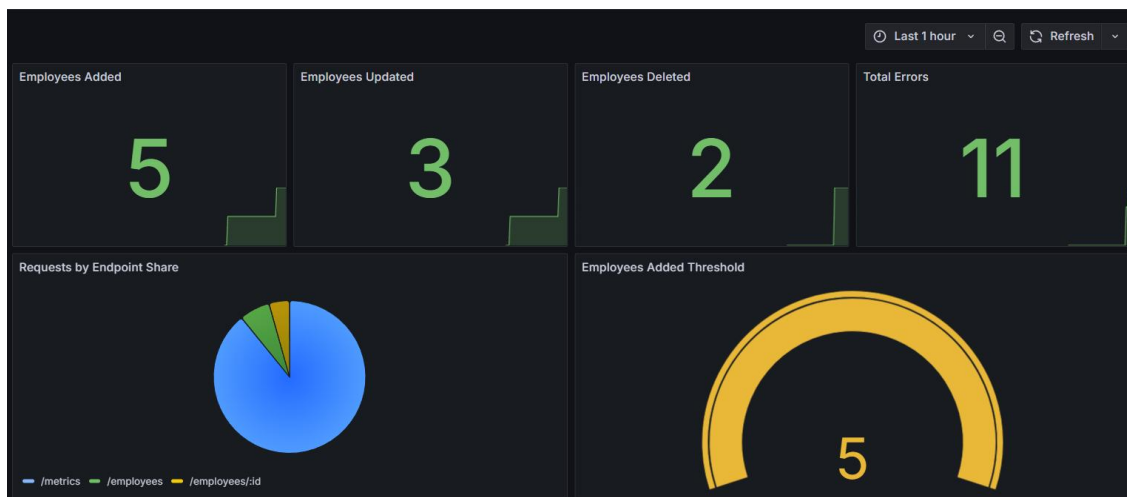
The Grafana dashboard includes the following visualization types:

- Stat Panels: Total employees added, updated, deleted, errors
- Time Series: HTTP requests over time, CPU usage, memory usage
- Bar Chart: Top request endpoints
- Gauge: Threshold tracking for employee additions
- Heatmap: Request duration distribution using histograms
- Table: Raw metric listings by endpoint or method

These visualizations allow for immediate and insightful understanding of application and system performance.

Why Grafana?

Grafana was chosen due to its strong integration with Prometheus and support for flexible, real-time visualizations. Its dashboard builder allows custom layouts, alert rules, and responsive interaction with data. Most importantly, it makes invisible metrics meaningful and actionable, helping developers and operators make informed decisions faster.



Stat Panels (Top Row):

These panels show the total number of employee-related operations:

- Employees Added: Value = 5
- Employees Updated: Value = 3
- Employees Deleted: Value = 2
- Total Errors: Value = 11

These values are drawn directly from Prometheus counters and are updated live. They provide a high-level overview of API usage and user behavior, as well as system robustness based on error frequency.

Pie Chart – Requests by Endpoint Share:

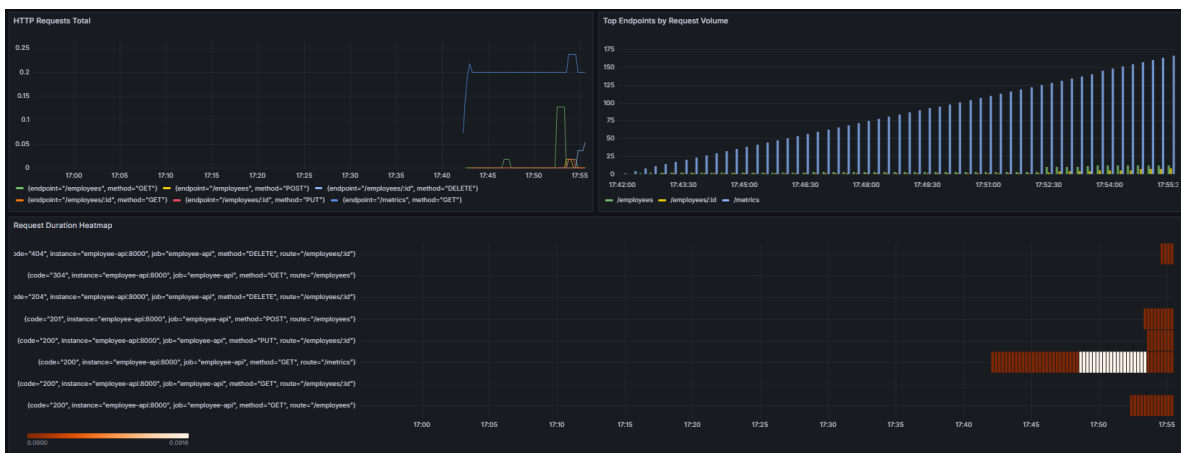
This chart shows the proportion of HTTP traffic distributed across different endpoints like /employees, /employees/:id, and /metrics.

Insight: It reveals which parts of the system are accessed most frequently, helping identify operational hot spots.

Gauge – Employees Added Threshold:

This gauge tracks the current number of employees added and visually compares it to a defined threshold.

Insight: It allows quick detection if employee creation activity becomes unusually high (which could trigger alerts).



This image displays panels for more technical monitoring:

Time Series – HTTP Requests Total:

This graph shows request rates over time, filtered by method and route.

Insight: Helps spot request spikes or sudden drops, which may correlate with system load, downtime, or misuse.

Bar Chart – Top Endpoints by Request Volume:

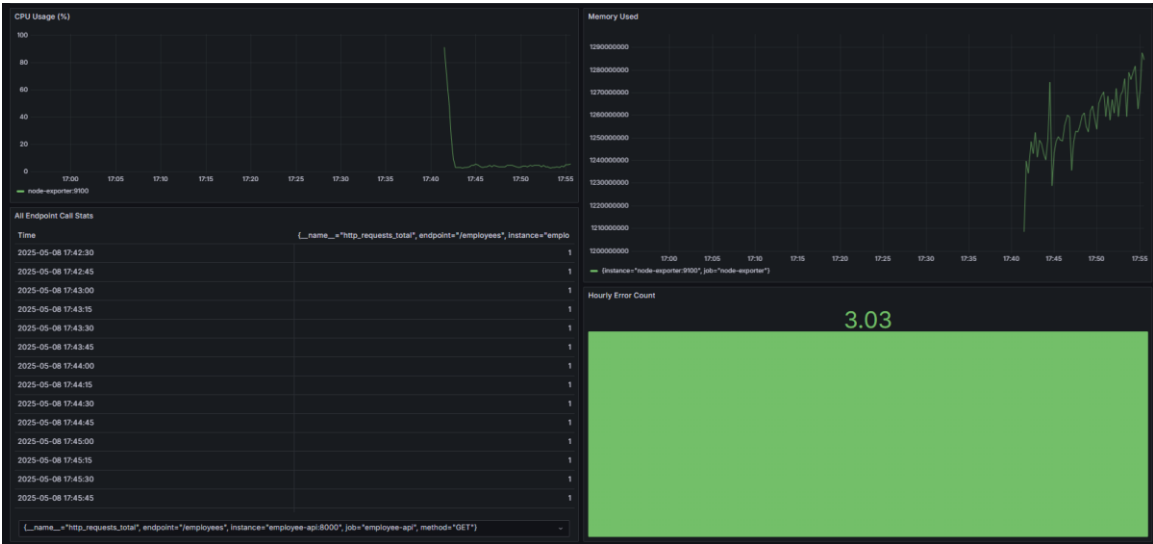
This panel shows the total number of requests per route using vertical bars.

Insight: It highlights the most frequently accessed API endpoints. In the image, /employees dominate the traffic, followed by /metrics.

Heatmap – Request Duration Distribution:

Using the histogram http_request_duration_seconds, this panel visualizes latency trends per route and HTTP method.

Insight: Helps identify routes that experience performance issues or unusually long processing times.



Focuses on system-level observability, showing data scraped by **node-exporter**:

CPU Usage (%):

Line graph showing CPU utilization over time.

Insight: A drop is visible, possibly after a burst of requests or a container restart.

Memory Used:

This shows increasing memory usage over time, useful for detecting memory leaks or under-provisioning.

Insight: A steady upward trend could indicate application inefficiencies.

Table – All Endpoint Call Stats:

A table listing timestamps and routes for every API call (http_requests_total by method and endpoint).

Insight: Ideal for request auditing or debugging activity patterns.

Stat Panel – Hourly Error Count:

This visual summarizes how many errors occurred in the last hour using `rate(errors_total[1h])`.

Insight: Provides an early warning of degrading API behavior.

5.4. Docker and Containerization

To ensure consistent, scalable, and portable deployment of the full system, all services were containerized using Docker and managed via a single Docker Compose file. This approach allows the entire observability pipeline from data generation to monitoring and alerting to run in isolated containers, eliminating dependency conflicts and simplifying deployment across different machines.

Services Defined in `docker-compose.yml` :

The `docker-compose.yml` file contains definitions for four services:

- 1. employee-api**

This is the main backend service, developed in Node.js. It is built using a custom Dockerfile, which installs dependencies, copies code files, and starts the Express server. It exposes REST endpoints (`/employees`, `/metrics`) and hosts application-specific Prometheus metrics.

- 2. prometheus**

This container runs the Prometheus server. It reads the configuration from `prometheus/prometheus.yml` and scrapes metrics from the API (`:8000`) and system exporter (`:9100`). It acts as a time-series database for metric storage and alert rule evaluation.

- 3. grafana**

Grafana handles the visualization of all collected metrics. A named volume is mapped to persist dashboard configurations, ensuring data and panel designs are not lost after restarts. Grafana runs on port 3000 and is configured to use Prometheus as its data source.

- 4. node-exporter**

This container is responsible for exposing system-level metrics such as CPU usage, memory usage, and disk activity. It runs on port 9100 and provides valuable infrastructure observability data.

Benefits of Docker:

The use of Docker provides numerous advantages:

- **Cross-Machine Compatibility:** Containers run the same way on any system with Docker installed, making testing and deployment easier.
- **Environment Consistency:** All dependencies (Node.js, Prometheus, Grafana) are included within containers, ensuring that no system-level setup is required.
- **No Manual Installation:** There's no need to install Prometheus or Grafana manually on the host, they're preconfigured in the Compose file.

- **Persistent Volumes:**

- A volume is defined to store **Grafana dashboards**, so visualizations are retained even if the container is rebuilt.
- The data.json file is also volume-mounted, ensuring that employee data and counter values are preserved across sessions.

Running and Managing the System:

To start the system with a fresh build (e.g., after code changes), the following command is used:

- `docker-compose up --build`

To stop and cleanly shut down all containers:

- `docker-compose down`

This setup ensures that development, testing, and demonstration environments are always clean, reproducible, and easy to restart.

The employee-api service is configured to write its operational state to a data.json file. This includes all employee records and Prometheus counter values. By mapping this file as a volume, its contents are retained even when the container is rebuilt or stopped. On server startup, the API reads this file and restores state ensuring the application can resume exactly where it left off.

```
app > Dockerfile > ...
24
25 # Use the official Node.js image from Docker Hub
26 FROM node:18
27
28 # Set working directory inside the container
29 WORKDIR /app
30
31 # Copy package.json and package-lock.json if available
32 COPY package*.json ./
33
34 # Install dependencies
35 RUN npm install
36
37 # Copy the entire app directory contents into the container
38 COPY . .
39
40 # Expose the API port
41 EXPOSE 8000
42
43 # Command to run your application
44 CMD ["node", "server.js"]
45
```

```
docker-compose.yml > ...
1
2
3 >Run All Services
services:
4   >Run Service
   employee-api:
5     build: ./app
6     container_name: employee-api
7     ports:
8       - "8000:8000"
9     volumes:
10      - ./app:/app
11
12   >Run Service
   prometheus:
13     image: prom/prometheus
14     container_name: prometheus
15     volumes:
16      - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
17     ports:
18      - "9090:9090"
19
20   >Run Service
   grafana:
21     image: grafana/grafana
22     container_name: grafana
23     ports:
24      - "3000:3000"
25     volumes:
26      - ./grafana:/etc/grafana/provisioning
27      - grafana-storage:/var/lib/grafana
28     depends_on:
29      - prometheus
30
31   >Run Service
   node-exporter:
32     image: prom/node-exporter
33     container_name: node-exporter
34     ports:
35      - "9100:9100"
36
37 volumes:
38   grafana-storage:
```


Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
vaued_assignment2	-	-	-	1.06%	21 minutes ago	
node-exporter	69e947992078	prom/node-exporter	9100:9100	0%	21 minutes ago	
employee-api	72de5aebce82	vaued_assignment2-employee-api	8000:8000	0.68%	21 minutes ago	
prometheus	44cfb7240b28	prom/prometheus	9090:9090	0%	21 minutes ago	
grafana	3c3dbb250ca9	grafana/grafana	3000:3000	0.38%	21 minutes ago	

5.5. Alerting Configuration

To complete the observability pipeline, Grafana's Unified Alerting system was configured to monitor critical application behaviors and notify the user when metric thresholds were exceeded. Alerts help ensure that system issues such as rapid changes or errors are not only visualized but also automatically tracked and responded to.

In this project, two key alerts were created using Prometheus metrics as the data source:

Alert 1: High Employee Addition Rate

This alert monitors how frequently new employees are being added to the system. It uses the PromQL expression:

$$\text{rate}(\text{employees_added_total}[1\text{m}]) > 5$$

This means that if more than 5 employees are added within one minute, the alert will enter a firing state. The alert is evaluated every minute and will remain active until the rate drops below the threshold. This is useful for detecting abnormal spikes in API usage or potential misuse of the system.

The screenshot displays the Grafana Alerting configuration for an alert named 'High Employee Addition Rate'. The alert is currently in a 'Normal' state. The configuration includes the following details:

- State:** Normal
- Name:** High Employee Addition Rate
- Health:** ok
- Summary:** (empty)
- Next evaluation:** in a minute
- Actions:** View, Edit, More
- Evaluate:** Every 1m
- Pending period:** 1m
- Keep firing for:** 1m
- Last evaluation:** a few seconds ago
- Evaluation time:** 5s
- Data source:** prometheus
- Instances:** 1 normal

The 'Instances' section shows a table with the following data:

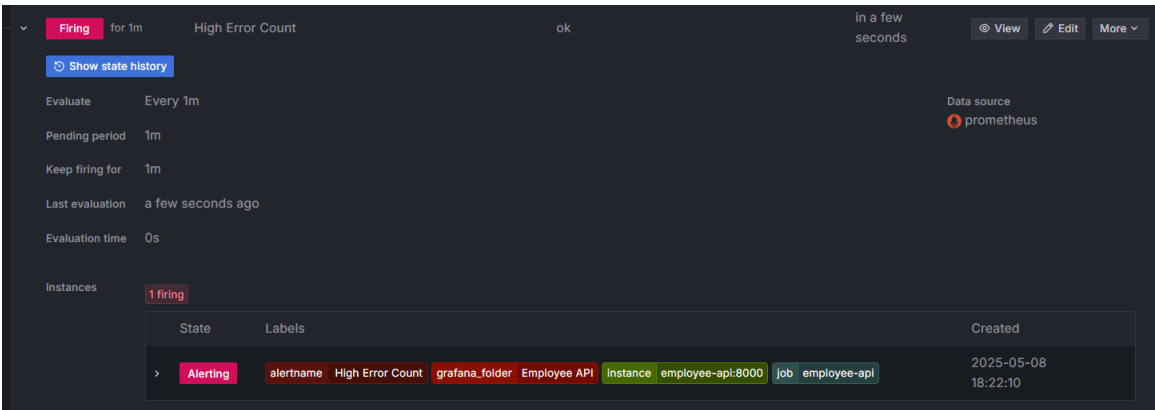
State	Labels	Created
Normal	alertname: High Employee Addition Rate, grafana_folder: Employee API, instance: employee-api:8000, job: employee-api	2025-05-08 18:21:10

Alert 2: High Error Count

This alert tracks the total number of operational errors (e.g., invalid requests, failed lookups). The expression used is:

`errors_total > 5`

This means that once the cumulative count of errors exceeds 5, the alert is triggered. This provides early warning for system issues, such as frequent user mistakes, broken client integrations, or bugs in API logic.



Alert Evaluation & Visibility

Both alerts were configured to evaluate every minute, ensuring near real-time responsiveness. The current state of each alert, such as OK, Pending, or Firing, is made visible in the Alerting tab of the Grafana UI. This centralizes monitoring and makes the system transparent and easy to manage without checking metrics manually.

The alerts also include labels and instance identifiers, which help identify which service (e.g., employee-api:8000) triggered the alert.

6. Task Breakdown

Member	Task	Responsibilities
IT21468360	API Developer & Metric Integration	<ul style="list-style-type: none">• Implement server.js with CRUD endpoints• Add Prometheus custom counters using prom-client• Expose /metrics endpoint• Setup data.json persistence logic
IT21489464	Prometheus & System Exporter Configuration	<ul style="list-style-type: none">• Configure Prometheus (prometheus.yml)• Define scrape targets (API and node-exporter)• Ensure scrape_interval and labels are correct• Test queries in Prometheus UI (rate(...), sum(...))
IT21924750	Grafana Dashboard & Visualizations	<ul style="list-style-type: none">• Set up Grafana interface and add Prometheus as a data source• Import or create dashboards with:<ul style="list-style-type: none">○ Stat panels○ Time series○ Gauges○ Heatmaps○ Tables and bar charts• Ensure panels are labeled and clean
IT21469046	Alerting + Docker Containerization	<ul style="list-style-type: none">• Create alert rules:<ul style="list-style-type: none">○ rate(employees_added_total[1m]) > 5○ errors_total > 5• Set alert conditions, evaluation periods, and summaries• Set up docker-compose.yml:<ul style="list-style-type: none">○ Define services: employee-api, prometheus, grafana, node-exporter○ Mount volumes for persistence• Handle Docker builds and re-deployment

7. Conclusion

The assignment was successfully completed with a fully functional, observable REST API for employee management. All three tasks were implemented: a working web service, real-time monitoring using Prometheus, rich Grafana dashboards, alert rules for automated observability, and containerized deployment for repeatability and portability.