RV College of Engineering®
Mysore Road, RV Vidyaniketan Post,
Bengaluru - 560059, Karnataka, India

Go, change the world®

DEPARTMENT OF

INFORMATION SCIENCE AND ENGINEERING

Innovative Experiment
Report On

"Metrics Collection and Monitoring Using Prometheus"

By

1RV23SSE10 RAVIKIRAN N S

1RV23SSE09 NAVASMEET NAYAK

1RV23SIT06 KSHITIZ SHARMA

Under the Guidance of

Prof. Rashmi R

Assistant Professor

Dept. of ISE

RV College of Engineering®

Course Name: API Development and Integration Lab

Course Code: MIT438L

SEPT 2023 - 24

Table of Contents

Metrics Collection and Monitoring Using Prometheus

INTRODUCTION

With the increasing complexity of web applications, it is critical to monitor their health and performance in real time. Monitoring tools like Prometheus provide a way to collect and analyze metrics, allowing developers to respond proactively to issues such as system failures or performance degradation.

This report focuses on implementing Prometheus in a MERN stack hotel booking application to monitor key performance indicators. The data collected will help developers maintain the application's stability, measure reliability, and optimize response times. Specific metrics monitored include Task Failure Rate, Error Rate, Mean Time Between Failures (MTBF), Mean Time to Failure (MTTF), and Mean Time to Recovery (MTTR).

SOFTWARE REQUIREMENTS WITH VERSION, INSTALLATION PROCEDURES

Following is the list of software requirements specified in order of installation:

Software Name	Version	Installation Link	Purpose
MongoDB	4.x or higher	MongoDB	NoSQL database for storing application data.
Express.js	4.17.x or compatible	Express.js	Backend framework for building RESTful APIs and microservices.
Node.js	16.x or higher	https://nodejs.org/en/download/prebuilt-installer/current	JavaScript runtime for running backend services.
React.js	17.x	React.js	Frontend library for creating user interfaces.
Prometheus	2.26.x or higher	https://prometheus.io/download/	Monitoring tool for collecting time-series data and system metrics.
Grafana	8.x or higher	https://grafana.com/grafana/download	Data visualization tool for creating dashboards to display Prometheus metrics.
Prometheus Client	prom-client 14.x	Prom Client	Node.js library for exposing custom metrics to Prometheus.
Insomnia	10.x or higher	https://insomnia.rest/download	API testing tool for verifying the endpoints and system responses.

SOURCE CODE LINK (GITHUB):

OVERVIEW OF THE ESSENTIAL ASPECTS OF THE API

The link below provides a clear and concise overview of the essential aspects of the created APIs, making it easy for users to understand and reuse:

The source code, including backend instrumentation for collecting and exposing metrics to Prometheus, can be accessed at the following **GitHub** repository:

- **GitHub Repository Link**: MERN-Stack-Metrics-Monitoring

The repository contains the full implementation, including the integration of Prometheus for monitoring system metrics, configuration files, and setup instructions.

LIST OF APIS WITH ITS PURPOSE

API Name	Method	Authentication	Purpose
Metrics API	GET	None	Exposes metrics to Prometheus via /metrics endpoint for system monitoring.
User API	POST, GET	Token	Handles user registration, login, and profile management.
Task Management API	POST, GET	Token	Manages task creation, updates, and status tracking.
Error Reporting API	POST	None	Logs system errors and application failures for Prometheus analysis.

DESCRIPTION ABOUT EACH MODULE

Overview of the MERN Stack Application

The MERN stack is a popular full-stack development framework consisting of:

- **MongoDB**: A NoSQL database used to store data in a flexible JSON-like format.
- **Express.js**: A web application framework for Node.js that simplifies the development of server-side logic.
- **React.js**: A frontend JavaScript library used for building interactive user interfaces.
- **Node.js**: A JavaScript runtime environment for executing server-side code.

In the context of this report, the MERN stack is used to build a hotel booking system where users can search, book rooms, and view their reservations. The backend (Node.js and Express) is critical for handling API requests, managing bookings, and interacting with MongoDB. Monitoring the performance of this backend is essential for ensuring the overall application's reliability and user satisfaction.

Prometheus: Metrics Collection and Monitoring

Prometheus is an open-source monitoring and alerting toolkit designed to collect time-series data. It operates on a pull-based model, where Prometheus periodically scrapes metrics from target applications through an exposed endpoint. The data collected is stored in a time-series database, where it can be queried and visualized to analyze trends and detect performance issues.

In this project, Prometheus was used to monitor the MERN application by collecting performance metrics from the backend. The application was instrumented using the prom-client library for Node.js to expose relevant metrics on a /metrics endpoint. These metrics were then scraped by Prometheus at regular intervals and stored for analysis.

Visualization with Grafana

For enhanced visualization of the collected metrics, Grafana was used. Grafana was configured to use Prometheus as a data source, and dashboards were created to display the performance metrics. Custom visualizations for Task Failure Rate, Error Rate, and MTTR helped track the application's health in real time.

IMPLEMENTATION DETAILS WITH TOOLS USED

Instrumenting the Application

The first step in collecting metrics was to instrument the Node.js application using the prom-client library. This allowed the application to expose metrics at a /metrics endpoint. Below is a code snippet showing how custom metrics were defined and registered:

javascript

```
const express = require('express');

const client = require('prom-client');


const app = express();

const register = new client.Registry();


// Define metrics

const taskFailureCounter = new client.Counter({ name: 'task_failure_count', help: 'Total number of failed tasks' });

const errorCounter = new client.Counter({ name: 'error_count', help: 'Total number of errors' });

const failureDuration = new client.Histogram({ name: 'failure_duration_seconds', help: 'Time between failures' });

const recoveryDuration = new client.Histogram({ name: 'recovery_duration_seconds', help: 'Time to recover from failure' });


// Register metrics

register.registerMetric(taskFailureCounter);

register.registerMetric(errorCounter);

register.registerMetric(failureDuration);

register.registerMetric(recoveryDuration);
```

```
// Expose /metrics endpoint
app.get('/metrics', async (req, res) => {
  res.setHeader('Content-Type', register.contentType);
  res.send(await register.metrics());
});

app.listen(3001, () => {
  console.log('App running on port 3001');
});
```

Configuring Prometheus

Prometheus was configured to scrape the /metrics endpoint of the application. This was achieved by adding a job configuration to the prometheus.yml file:

yaml

```
scrape_configs:
  - job_name: 'mern_application'
    static_configs:
      - targets: ['localhost:3001']  # Application's metrics endpoint
```

Prometheus then periodically queried this endpoint to collect and store the exposed metrics.

Testing and Validation

After deploying the instrumented application and configuring Prometheus, the /metrics endpoint was tested to ensure proper exposure of metrics. This was done by visiting *http://localhost:3001/metrics or using curl:*

```
curl http://localhost:3001/metrics
```

Prometheus was able to successfully scrape the metrics, and queries were tested via Prometheus' web interface.

Metrics Analysis with PromQL

Using **PromQL** (Prometheus Query Language), the collected metrics were analyzed. Some key queries used for the analysis include:

- **Task Failure Rate**: rate(task_failure_count[5m])
- **Error Rate**: rate(error_count[5m])
- **MTBF (Mean Time Between Failures)**: histogram_quantile(0.95,sum(rate(failure_duration_seconds_bucket[5m])) by (le))
- **MTTF (Mean Time to Failure)**: histogram_quantile(0.95,sum(rate(failure_duration_seconds_bucket[5m])) by (le))
- **MTTR (Mean Time to Recovery)**: histogram_quantile(0.95,sum(rate(recovery_duration_seconds_bucket[5m])) by (le))

These queries provided insights into system performance trends and helped visualize the system's health over time.

WORKING PROCEDURE

Prometheus: Metrics Collection and Monitoring

Prometheus is an open-source monitoring and alerting toolkit designed to collect time-series data. It operates on a pull-based model, where Prometheus periodically scrapes metrics from target applications through an exposed endpoint. The data collected is stored in a time-series database, where it can be queried and visualized to analyze trends and detect performance issues.

In this project, Prometheus was used to monitor the MERN application by collecting performance metrics from the backend. The application was instrumented using the prom-client library for Node.js to expose relevant metrics on a /metrics endpoint. These metrics were then scraped by Prometheus at regular intervals and stored for analysis.

Key Metrics Collected

Task Failure Rate

Task Failure Rate refers to the frequency at which tasks (such as booking operations) fail. This metric is important because frequent task failures can indicate bugs, misconfigurations, or external system issues (such as a database going offline). A **Counter** in Prometheus was used to track this metric by incrementing each time a task fails.

Error Rate

Error Rate tracks the number of errors encountered by the backend during request handling. Errors could result from failed database connections, invalid user inputs, or network issues. Tracking this metric allows developers to identify patterns and respond to recurring problems quickly. Like Task Failure Rate, this was implemented using a **Counter** that increments on each error event.

Mean Time Between Failures (MTBF)

Mean Time Between Failures (MTBF) measures the average amount of time between consecutive failures. This metric is critical for understanding how frequently the system fails and can indicate overall system reliability. In Prometheus, **Histograms** were used to measure the duration between failures, allowing for the calculation of MTBF.

Mean Time to Failure (MTTF)

Mean Time to Failure (MTTF) is a reliability metric that shows the average time the system operates before experiencing a failure. MTTF gives insight into how long the system can function reliably before issues arise. Prometheus collects this data using **Histograms**, where the time to the next failure is recorded each time a failure occurs.

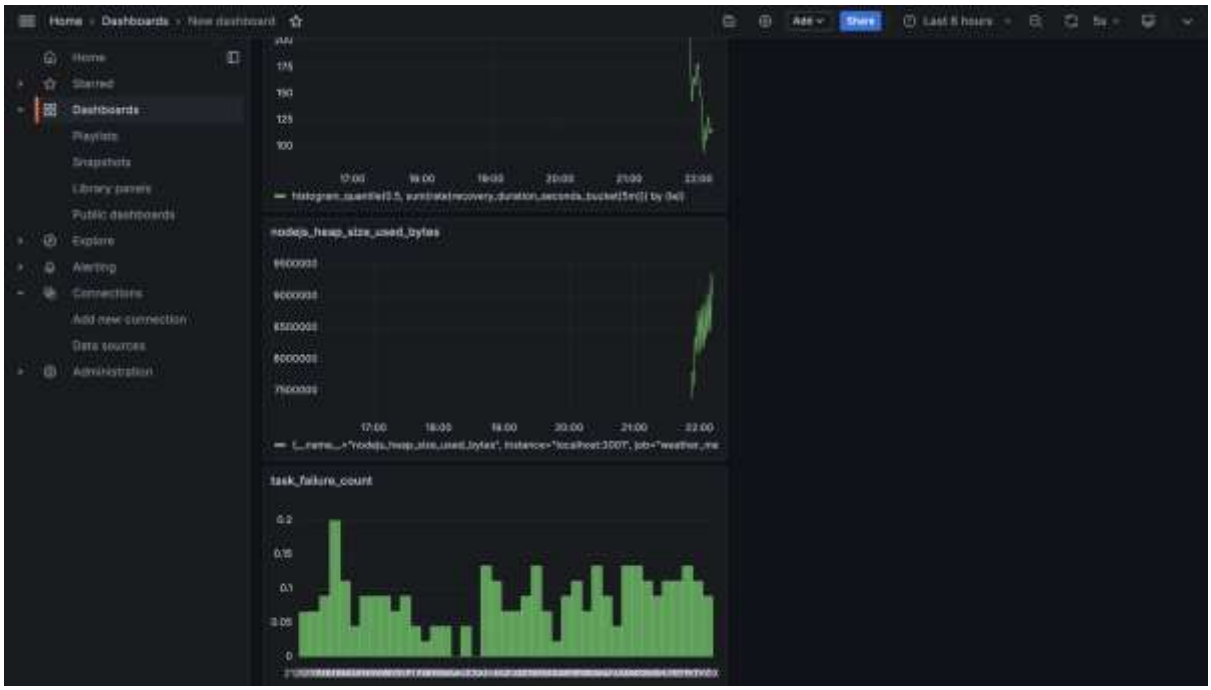
Mean Time to Recovery (MTTR)

Mean Time to Recovery (MTTR) measures how long it takes to recover from a failure. The shorter the MTTR, the faster the system can return to normal operations after an issue is detected. This metric is important for assessing how well the application handles failures and how quickly recovery procedures are executed. **Histograms** in Prometheus were used to log recovery times and calculate MTTR

SCREENSHOTS

Visualization with Grafana

For enhanced visualization of the collected metrics, Grafana was used. Grafana was configured to use Prometheus as a data source, and dashboards were created to display the performance metrics. Custom visualizations for Task Failure Rate, Error Rate, and MTTR helped track the application's health in real time.




Home > Dashboards > New dashboard > View panel
Share
Last 6 hours
Home
Starred
Dashboards
Playlists
Snapshots
Library panels
Public dashboards
Explore
Alerting
Connections
Add new connection
Data sources
Administration
task_failure_count
0.22
0.2
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0

CONCLUSION

Integrating Prometheus with the MERN stack hotel booking application has proven to be an effective solution for monitoring critical performance metrics. Prometheus provided a robust mechanism for tracking key metrics such as Task Failure Rate, Error Rate, Mean Time Between Failures (MTBF), Mean Time To Failure (MTTF), and Mean Time To Repair (MTTR). These metrics enabled continuous evaluation of the application's reliability and performance, offering valuable insights into system behavior. The integration allowed real-time monitoring of system health, empowering the development and operations teams to detect potential issues early and respond swiftly to any performance degradation or failures. This proactive approach resulted in reduced downtime and improved overall application stability, ensuring a better user experience for customers of the hotel booking platform.