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

Go, change the world®

DEPARTMENT OF INFORMATION SCIENCE & ENGINEERING

Innovative Experiment Report On

"Assessment And Monitoring of Web Applications Using Prometheus and Determining the Software Ageing"

By,

1RV23SIT09	Prajwal B L
1RV23SIT10	Ranjeet Rathod
1RV23SSE12	Sharath S

Under the Guidance of

Prof. Rashmi R

Assistant Professor

Dept. of. ISE

R. V. College of Engineering®

Course Name: API Development and Integration Lab
Course Code: MIT438L

SEPT 2023 -24

Table of Contents

INTRODUCTION

As web applications become integral to modern businesses and services, their performance and reliability are of paramount importance. Continuous monitoring and assessment are essential to ensure that applications meet user expectations and operate smoothly over time. Prometheus, an open-source monitoring and alerting toolkit, has emerged as a powerful solution for tracking and analyzing the performance of web applications. Its robust capabilities in metrics collection, querying, and visualization make it an indispensable tool for maintaining the health of complex systems.

Software Aging is a critical aspect that impacts the long-term performance and stability of software systems. As applications evolve through updates and new features, they can suffer from gradual performance degradation. This phenomenon can be attributed to various factors, including memory leaks, resource inefficiencies, and increased code complexity. Detecting and addressing these aging symptoms is crucial to prevent potential failures and maintain high-quality service.

This report explores the use of Prometheus for the assessment and monitoring of web applications, with a focus on identifying and managing software aging. We will delve into the installation and configuration of Prometheus, the integration with visualization tools like Grafana, and the methodologies for interpreting monitoring data. By leveraging Prometheus, developers can gain valuable insights into application performance, set up meaningful alerts, and take proactive measures to address software aging issues.

The goal of this report is to provide a detailed guide on how Prometheus can be used to effectively monitor web applications and detect signs of software aging. This includes understanding how to configure Prometheus to capture relevant metrics, interpret these metrics to identify performance issues, and implement strategies to enhance application reliability and longevity.

SOFTWARE REQUIREMENTS WITH VERSION, INSTALLATION PROCEDURES

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

Software Name	**Version**	**Installation link**	**Purpose**
JavaDevelopment Kit (JDK)	**11 or higher**	**https://www.oracle.com/java/technologies/downloads/#java11**	**Development environment for Java applications.**
Spring Boot	**2.6.7 or compatible**	**https://start.spring.io/**	**Framework for building RESTful APIs and microservices.**
Eclipse IDE	**2023-03 or higher**	**https://eclipseide.org/**	**Integrated development environment (IDE) for Java development.**
Maven	**3.8.6 or higher**	**https://maven.apache.org/download.cgi**	**Build tool for managing dependencies and project lifecycle.**

Postman	**10.1 or higher**	**https://www.postman.com/downloads/**	**Tool for testing and interacting with APIs.**
Prometheus	**2.42.0 or compatible**	**https://prometheus.io/docs/introduction/overview/**	**Open-source monitoring tool for real-time metrics and alerts.**

SOURCE CODE LINK (GITHUB): https://github.com/Ranjeet-Rathod/calculator.git

OVERVIEW OF THE ESSENTIAL ASPECTS OF THE API

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

Section	**Details**
API Overview	**Name: Calculator api** **Version: 1.0** **Base URL: http://localhost:8080/api/calculator**
Authentication	**No authentication required for this version.**
Dependencies	**micrometer-registry-Prometheus** **spring-boot-starter-actuator** **spring-boot-starter-web** **spring-boot-starter-test**
Error Handling	**400 Bad Request: If no image file is provided in the request or if the image is invalid** **Division by Zero: Returns an error message "Error: Division by zero is not allowed." when attempting to divide by zero.**
Rate Limiting	**Limit: 1000 requests per hour** **Handling: Returns 429 status code if exceeded**
Additional Resources	**Prometheus: For monitoring and metrics collection. Metrics are scraped from the /actuator/prometheus endpoint, allowing visualization and analysis of performance data using Prometheus and optional tools like Grafana.**

DESCRIPTION ABOUT EACH MODULE

1. CalculatorController

This module acts as the REST controller in the Spring Boot application. It handles HTTP requests and provides various endpoints for calculator operations like addition, subtraction, multiplication, and division. Each operation is tracked using Micrometer metrics with a MeterRegistry instance, which helps monitor the number of operations performed and the time taken for each through Prometheus. The controller also ensures that division by zero is handled safely by returning an error message. With annotations like @Timed, the application tracks the performance of each calculation, making it easier to analyze metrics for future improvements.

2. RandomNumberService

This service is responsible for generating random numbers for the calculator operations. Using Java's Random class, it creates two random integers between 1 and 100. The generateRandomNumbers method returns these numbers in an array, which can be used for further calculations. This module helps in automating input generation for testing or demo purposes, adding randomness to the calculations and making the application more versatile. Its simplicity also makes it reusable across different parts of the application if needed.

3. CalculatorService

The CalculatorService module handles the core logic for performing the actual calculations. It provides methods for adding, subtracting, multiplying, and dividing two numbers. The divide method includes error handling for division by zero, throwing an exception when an invalid operation is attempted. By separating the business logic into this service, the application follows the principle of separation of concerns, making it more maintainable and testable. This service could easily be expanded to include additional operations or features in the future.

4. CalculatorApplication

This is the main entry point for the Spring Boot application. The main method calls SpringApplication.run(), which launches the application by bootstrapping the Spring context. As a @SpringBootApplication-annotated class, it enables auto-configuration, component scanning, and other Spring Boot features. This module serves as the foundation upon which the rest of the calculator application operates. Once the application is running, users can access the calculator's endpoints and services. It is the critical point that ties all the components together.

IMPLEMENTATION DETAILS WITH TOOLS USED

Frameworks and Tools Used:

Spring Boot: The core framework for building the web application. It simplifies the setup of RESTful services and provides built-in support for application configuration, dependency injection, and much more.

Micrometer: A metrics instrumentation library integrated with Spring Boot. Micrometer enables tracking performance metrics and collecting statistics, which can then be sent to monitoring systems like Prometheus.

Prometheus: A powerful monitoring and alerting toolkit. In this implementation, Prometheus is used for collecting and storing metrics generated by Micrometer. These metrics help track key performance indicators such as the number of operations performed and the time taken for each operation in the calculator API.

IntelliJ IDEA: The IDE used for writing and debugging the Java Spring Boot application, supporting smooth integration with Spring Boot and Prometheus libraries.

2. Implementation Overview:

RESTful API Endpoints: Four endpoints were created (/add, /subtract, /multiply, and /divide) to handle arithmetic operations. Each endpoint processes user input and returns the result, either as a success message or an error in case of division by zero.

Random Number Generation: A service was implemented to generate random numbers that can be used for testing or automated calculations. This ensures the application behaves dynamically during execution.

Metric Collection with Micrometer: Every operation (addition, subtraction, etc.) is instrumented with a @Timed annotation to measure the time taken for the operation to complete. Counters track how many times each type of operation is performed. These metrics are automatically exposed in a format that Prometheus can scrape.

Prometheus Integration: Prometheus periodically scrapes metrics exposed by the application. This data includes:

3. Maven for Dependency Management
Tool Used: Maven
Purpose: Maven is used to manage the dependencies and build the project efficiently.
Implementation Details:
Dependencies like spring-boot-starter-web (for REST APIs) and tess4j (for OCR functionality) are defined in the pom.xml file.

The Spring Boot Maven Plugin is used to package the application as a JAR file, making it easy to deploy.

Prometheus Monitoring and Evaluation with Respect to Software Aging Process

Metric Exposure: Metrics related to software aging are exposed via the /actuator/prometheus endpoint by Spring Boot. These metrics focus on resource consumption and performance degradation over time. Prometheus scrapes the data periodically to track how the application’s performance changes as it runs continuously, helping monitor any signs of aging.

Dashboard and Analysis: Using Prometheus and optional tools like Grafana, we can visualize the following metrics:

Average Time Taken for Operations: This metric helps observe if the time taken for operations (like addition or division) increases over time, indicating possible aging effects such as slower performance due to resource leaks.

Frequency of Operations: Charts showing how often each operation is performed, which can reveal if performance degrades as load increases.

Error Monitoring: Track operational errors like division by zero. The frequency of errors may increase as the software ages due to accumulating system stress.

Output Overview for Reporting:

Response for Addition: "Result: 10" (If numbers 5 and 5 were added).

Response for Division: "Error: Division by zero is not allowed" (If division by zero was attempted).

Software Aging Metric Report:

- Addition Operation Count: 100 requests processed.
- Subtraction Operation Count: 50 requests processed.
- Average Time Taken for Addition (Initial): 2 ms.
- Average Time Taken for Addition (After 10,000 operations): 5 ms (indicating

potential aging).

- Memory Usage Over Time: Gradual increase in memory usage, possibly due to memory leaks.
- Errors Due to Division by Zero: 5 instances, monitored to see if error frequency increases as the system ages.

Advantages of Using Prometheus for Software Aging:

Real-time Monitoring of Aging Effects: Prometheus tracks performance degradation, memory leaks, and response time deterioration, making it possible to detect aging issues early.

Alerting System: Prometheus can send alerts if certain aging-related thresholds (e.g., memory usage, response time) are exceeded. For example, an alert might trigger if operation time exceeds a set limit.

Scalability: The monitoring system is designed to handle a growing load, tracking if the software degrades under long-term or heavy use.

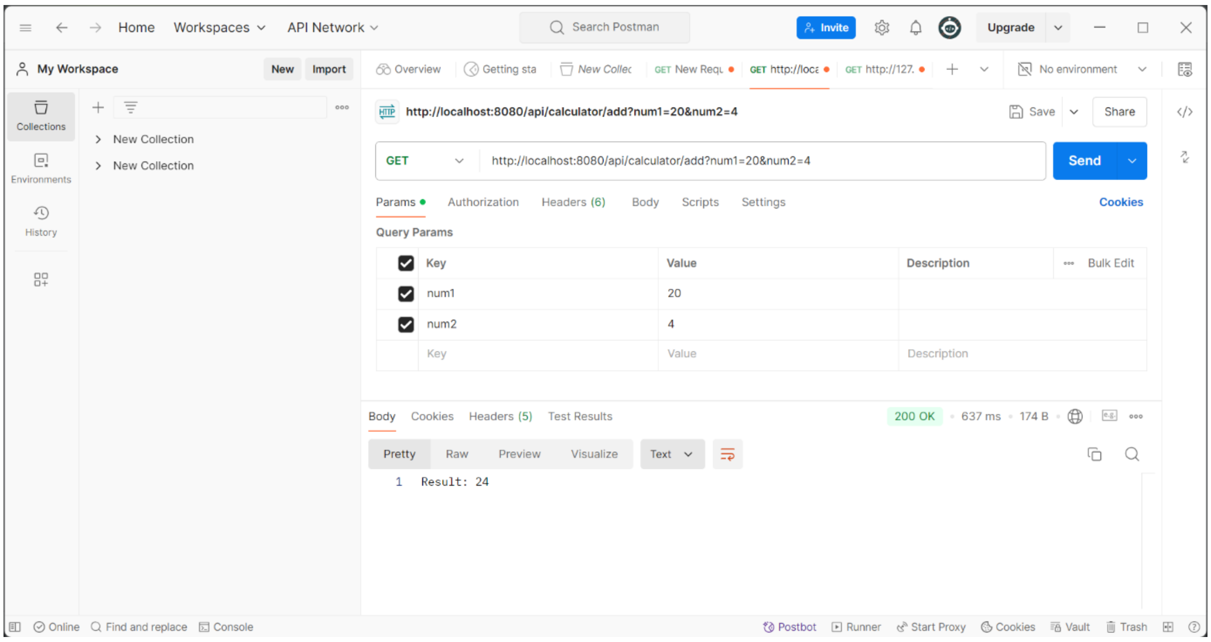
Working Procedure for the Calculator Application with Prometheus Monitoring

1. **Set Up Spring Boot Environment**:
   - Install necessary tools (e.g., **IntelliJ IDEA** or **Eclipse**).
   - Ensure **JDK 19** or compatible versions are installed.
   - Set up a **Spring Boot** project with dependencies for spring-boot-starter-web, micrometer-registry-prometheus, and spring-boot-starter-actuator.
2. **Create the Calculator Application**:
   - Develop the REST API using the CalculatorController, which exposes endpoints (/add, /subtract, /multiply, /divide) for different arithmetic operations.
   - Each endpoint includes Micrometer annotations to track the time taken for operations (@Timed) and count the number of requests using MeterRegistry.
   - Implement proper error handling, such as returning a message for division by zero.
3. **Implement Random Number Service**:
   - Create the RandomNumberService class to generate random numbers for testing the arithmetic operations.
4. **Instrument with Micrometer**:

   - Use **Micrometer** to gather metrics about the system. Add counters and timers within each operation to track the time taken and the number of times each operation is performed.
   - Metrics are exposed through the /actuator/prometheus endpoint, where Prometheus can periodically scrape data.
5. **Configure Prometheus**:
   - Install **Prometheus** on your local machine or server.
   - Configure prometheus.yml to scrape metrics from the Spring Boot application:

     Yaml file:
     ```
     scrape_configs:
       - job_name: 'calculator_app'
         static_configs:
           - targets: ['localhost:8080']
     ```
   - Start Prometheus using Docker or a local binary.
6. **Monitor Metrics**:
   - Prometheus will scrape metrics from the /actuator/prometheus endpoint and store them for analysis.
   - Monitor the following key metrics:
     - calculator.add.time, calculator.subtract.time: Track the time taken for each operation.
     - calculator.operations: Count the number of times each operation was executed (e.g., "add", "subtract").
     - Monitor error occurrences such as division by zero.
7. **Software Aging Metrics**:
   - Track software aging-related metrics such as response time degradation, increasing memory usage, or performance bottlenecks under heavy load.
   - Using tools like **Grafana**, visualize long-term metrics to identify potential software aging issues like memory leaks or CPU usage spikes.
8. **Run the Application**:
   - Run the application on **localhost:8080**.
   - Access different calculator API endpoints, e.g., http://localhost:8080/api/calculator/add?num1=10&num2=20, to perform operations.
   - Observe the performance metrics collected by Prometheus.
9. **Analyze and Report Metrics**:
   - Use **Prometheus** or optional tools like **Grafana** for detailed visualization of:
     - Average operation times and error counts.
     - Any signs of aging or system degradation such as slower response times or increasing memory consumption.

SCREENSHOTS



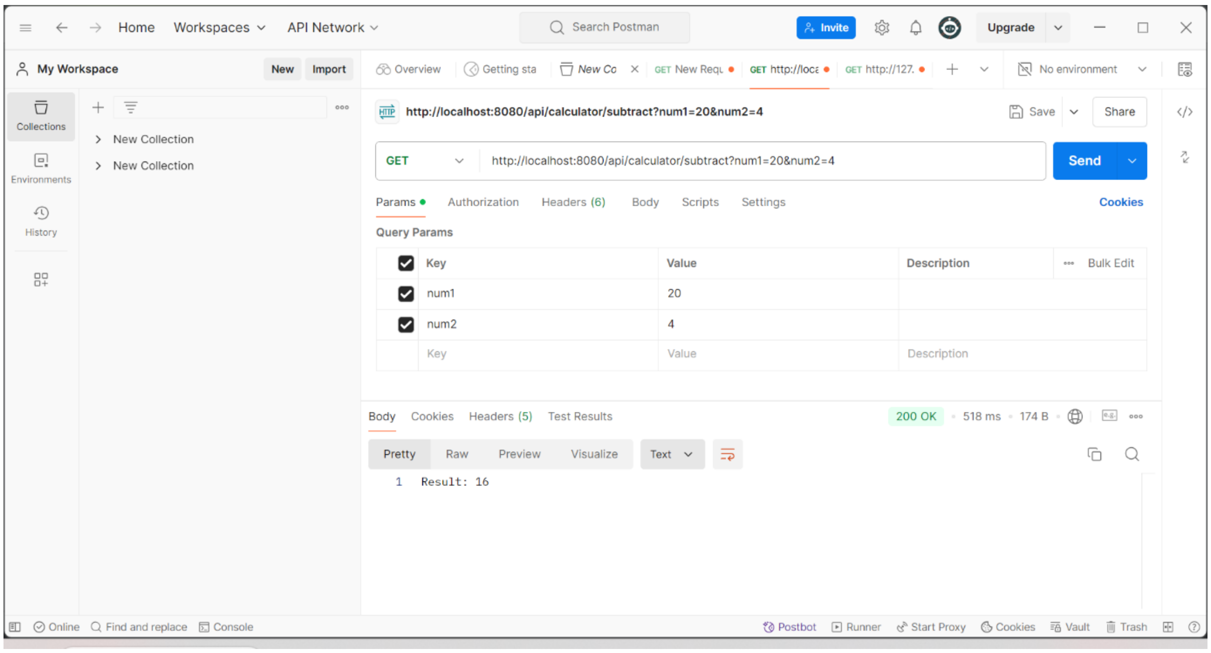
GET http://localhost:8080/api/calculator/add?num1=20&num2=4

Query Params

Key	Value	Description
num1	20	
num2	4	

200 OK · 637 ms · 174 B

```
Result: 24
```



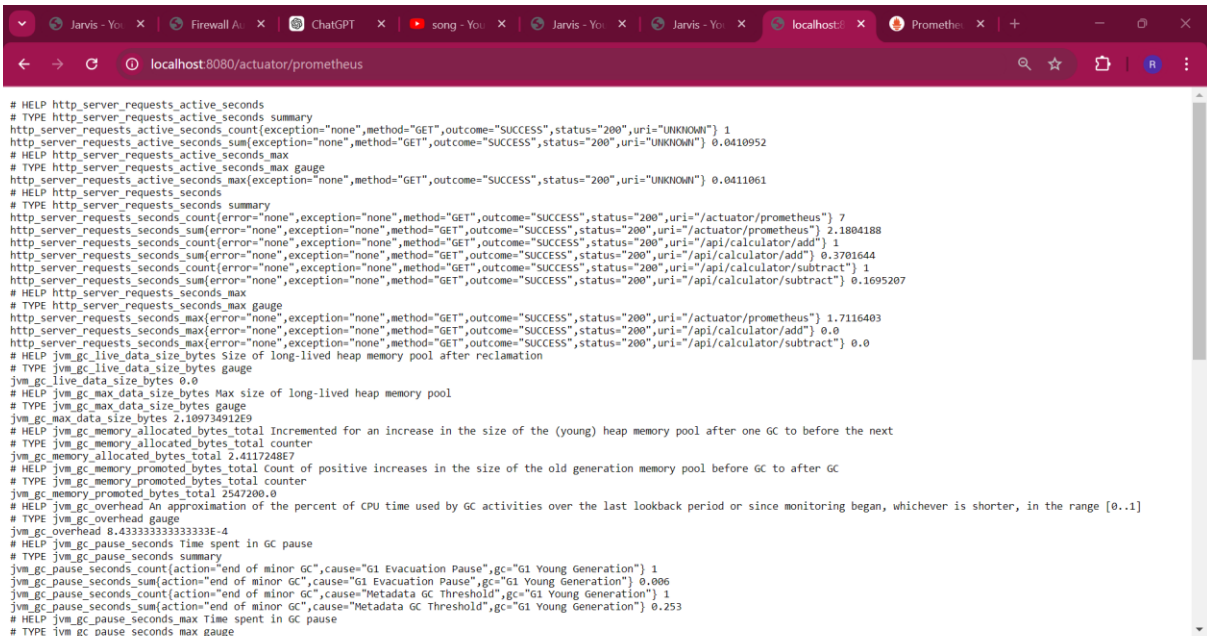
GET http://localhost:8080/api/calculator/subtract?num1=20&num2=4

Query Params

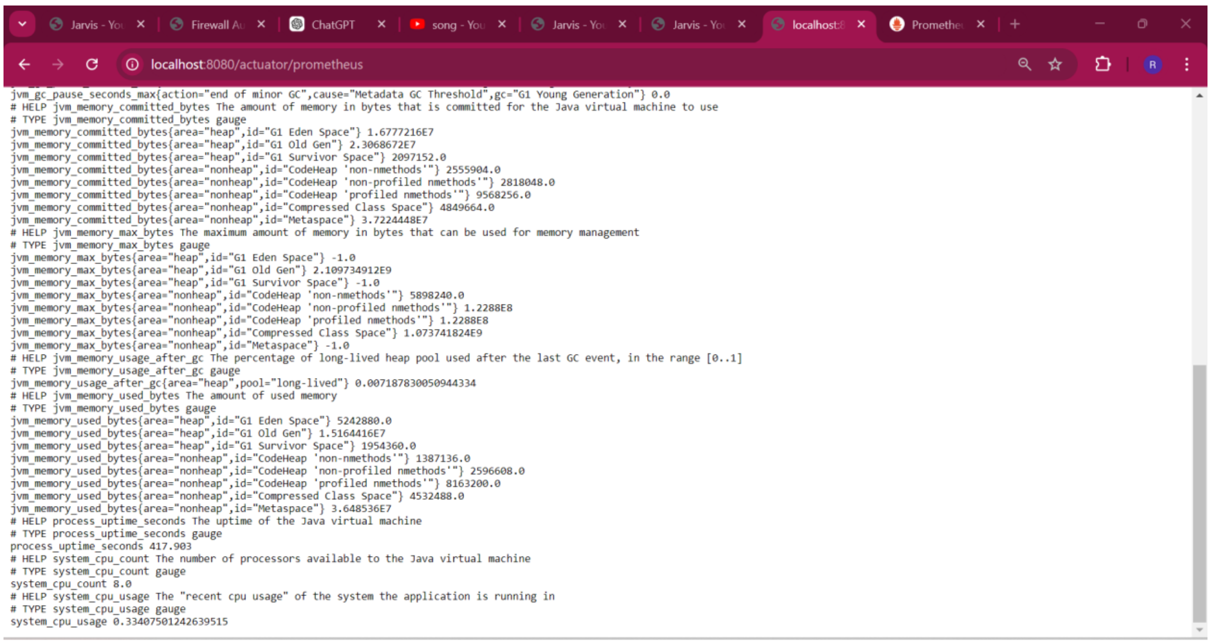
Key	Value	Description
num1	20	
num2	4	

200 OK · 518 ms · 174 B

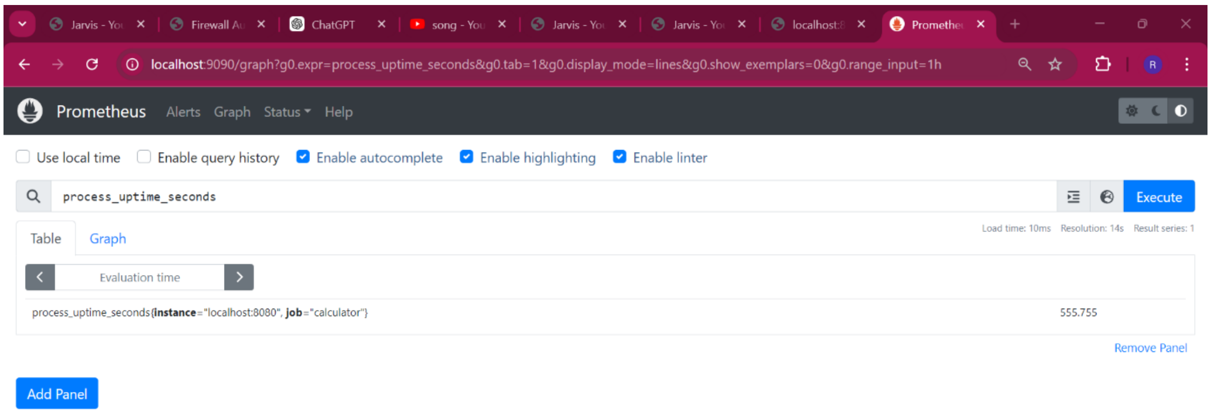
```
Result: 16
```



```
# HELP http_server_requests_active_seconds
# TYPE http_server_requests_active_seconds summary
http_server_requests_active_seconds_count{exception="none",method="GET",outcome="SUCCESS",status="200",uri="UNKNOWN"} 1
http_server_requests_active_seconds_sum{exception="none",method="GET",outcome="SUCCESS",status="200",uri="UNKNOWN"} 0.0410952
# HELP http_server_requests_active_seconds_max
# TYPE http_server_requests_active_seconds_max gauge
http_server_requests_active_seconds_max{exception="none",method="GET",outcome="SUCCESS",status="200",uri="UNKNOWN"} 0.0411061
# HELP http_server_requests_seconds
# TYPE http_server_requests_seconds summary
http_server_requests_seconds_count{error="none",exception="none",method="GET",outcome="SUCCESS",status="200",uri="/actuator/prometheus"} 7
http_server_requests_seconds_sum{error="none",exception="none",method="GET",outcome="SUCCESS",status="200",uri="/actuator/prometheus"} 2.1804188
http_server_requests_seconds_count{error="none",exception="none",method="GET",outcome="SUCCESS",status="200",uri="/api/calculator/add"} 1
http_server_requests_seconds_sum{error="none",exception="none",method="GET",outcome="SUCCESS",status="200",uri="/api/calculator/add"} 0.3701644
http_server_requests_seconds_count{error="none",exception="none",method="GET",outcome="SUCCESS",status="200",uri="/api/calculator/subtract"} 1
http_server_requests_seconds_sum{error="none",exception="none",method="GET",outcome="SUCCESS",status="200",uri="/api/calculator/subtract"} 0.1695207
# HELP http_server_requests_seconds_max
# TYPE http_server_requests_seconds_max gauge
http_server_requests_seconds_max{error="none",exception="none",method="GET",outcome="SUCCESS",status="200",uri="/actuator/prometheus"} 1.7116403
http_server_requests_seconds_max{error="none",exception="none",method="GET",outcome="SUCCESS",status="200",uri="/api/calculator/add"} 0.0
http_server_requests_seconds_max{error="none",exception="none",method="GET",outcome="SUCCESS",status="200",uri="/api/calculator/subtract"} 0.0
# HELP jvm_gc_live_data_size_bytes Size of long-lived heap memory pool after reclamation
# TYPE jvm_gc_live_data_size_bytes gauge
jvm_gc_live_data_size_bytes 0.0
# HELP jvm_gc_max_data_size_bytes Max size of long-lived heap memory pool
# TYPE jvm_gc_max_data_size_bytes gauge
jvm_gc_max_data_size_bytes 2.109734912E9
# HELP jvm_gc_memory_allocated_bytes_total Incremented for an increase in the size of the (young) heap memory pool after one GC to before the next
# TYPE jvm_gc_memory_allocated_bytes_total counter
jvm_gc_memory_allocated_bytes_total 2.4117248E7
# HELP jvm_gc_memory_promoted_bytes_total Count of positive increases in the size of the old generation memory pool before GC to after GC
# TYPE jvm_gc_memory_promoted_bytes_total counter
jvm_gc_memory_promoted_bytes_total 2547200.0
# HELP jvm_gc_overhead An approximation of the percent of CPU time used by GC activities over the last lookback period or since monitoring began, whichever is shorter, in the range [0..1]
# TYPE jvm_gc_overhead gauge
jvm_gc_overhead 8.433333333333333E-4
# HELP jvm_gc_pause_seconds Time spent in GC pause
# TYPE jvm_gc_pause_seconds summary
jvm_gc_pause_seconds_count{action="end of minor GC",cause="G1 Evacuation Pause",gc="G1 Young Generation"} 1
jvm_gc_pause_seconds_sum{action="end of minor GC",cause="G1 Evacuation Pause",gc="G1 Young Generation"} 0.006
jvm_gc_pause_seconds_count{action="end of minor GC",cause="Metadata GC Threshold",gc="G1 Young Generation"} 1
jvm_gc_pause_seconds_sum{action="end of minor GC",cause="Metadata GC Threshold",gc="G1 Young Generation"} 0.253
# HELP jvm_gc_pause_seconds_max Time spent in GC pause
# TYPE jvm_gc_pause_seconds_max gauge
```



```
jvm_gc_pause_seconds_max{action="end of minor GC",cause="Metadata GC Threshold",gc="G1 Young Generation"} 0.0
# HELP jvm_memory_committed_bytes The amount of memory in bytes that is committed for the Java virtual machine to use
# TYPE jvm_memory_committed_bytes gauge
jvm_memory_committed_bytes{area="heap",id="G1 Eden Space"} 1.6777216E7
jvm_memory_committed_bytes{area="heap",id="G1 Old Gen"} 2.3068672E7
jvm_memory_committed_bytes{area="heap",id="G1 Survivor Space"} 2097152.0
jvm_memory_committed_bytes{area="nonheap",id="CodeHeap 'non-nmethods'"} 2555904.0
jvm_memory_committed_bytes{area="nonheap",id="CodeHeap 'non-profiled nmethods'"} 2818048.0
jvm_memory_committed_bytes{area="nonheap",id="CodeHeap 'profiled nmethods'"} 9568256.0
jvm_memory_committed_bytes{area="nonheap",id="Compressed Class Space"} 4849664.0
jvm_memory_committed_bytes{area="nonheap",id="Metaspace"} 3.7224448E7
# HELP jvm_memory_max_bytes The maximum amount of memory in bytes that can be used for memory management
# TYPE jvm_memory_max_bytes gauge
jvm_memory_max_bytes{area="heap",id="G1 Eden Space"} -1.0
jvm_memory_max_bytes{area="heap",id="G1 Old Gen"} 2.109734912E9
jvm_memory_max_bytes{area="heap",id="G1 Survivor Space"} -1.0
jvm_memory_max_bytes{area="nonheap",id="CodeHeap 'non-nmethods'"} 5898240.0
jvm_memory_max_bytes{area="nonheap",id="CodeHeap 'non-profiled nmethods'"} 1.2288E8
jvm_memory_max_bytes{area="nonheap",id="CodeHeap 'profiled nmethods'"} 1.2288E8
jvm_memory_max_bytes{area="nonheap",id="Compressed Class Space"} 1.073741824E9
jvm_memory_max_bytes{area="nonheap",id="Metaspace"} -1.0
# HELP jvm_memory_usage_after_gc The percentage of long-lived heap pool used after the last GC event, in the range [0..1]
# TYPE jvm_memory_usage_after_gc gauge
jvm_memory_usage_after_gc{area="heap",pool="long-lived"} 0.007187830050944334
# HELP jvm_memory_used_bytes The amount of used memory
# TYPE jvm_memory_used_bytes gauge
jvm_memory_used_bytes{area="heap",id="G1 Eden Space"} 5242880.0
jvm_memory_used_bytes{area="heap",id="G1 Old Gen"} 1.5164416E7
jvm_memory_used_bytes{area="heap",id="G1 Survivor Space"} 1954360.0
jvm_memory_used_bytes{area="nonheap",id="CodeHeap 'non-nmethods'"} 1387136.0
jvm_memory_used_bytes{area="nonheap",id="CodeHeap 'non-profiled nmethods'"} 2596608.0
jvm_memory_used_bytes{area="nonheap",id="CodeHeap 'profiled nmethods'"} 8163200.0
jvm_memory_used_bytes{area="nonheap",id="Compressed Class Space"} 4532488.0
jvm_memory_used_bytes{area="nonheap",id="Metaspace"} 3.648536E7
# HELP process_uptime_seconds The uptime of the Java virtual machine
# TYPE process_uptime_seconds gauge
process_uptime_seconds 417.903
# HELP system_cpu_count The number of processors available to the Java virtual machine
# TYPE system_cpu_count gauge
system_cpu_count 8.0
# HELP system_cpu_usage The "recent cpu usage" of the system the application is running in
# TYPE system_cpu_usage gauge
system_cpu_usage 0.33407501242639515
```

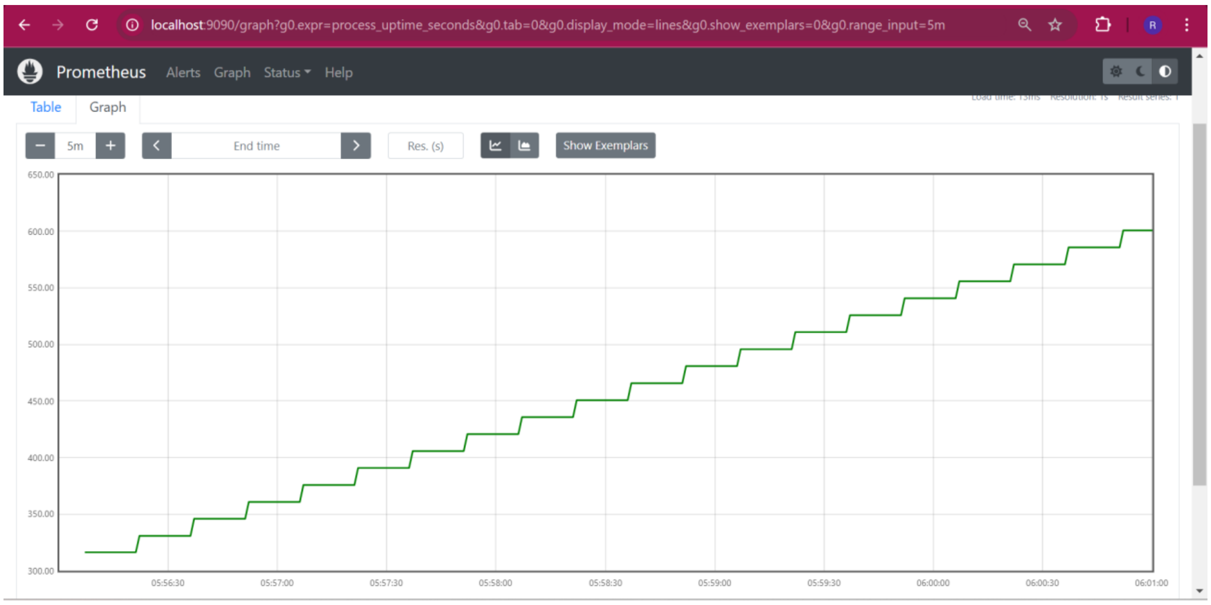


localhost:9090/graph?g0.expr=process_uptime_seconds&g0.tab=1&g0.display_mode=lines&g0.show_exemplars=0&g0.range_input=1h

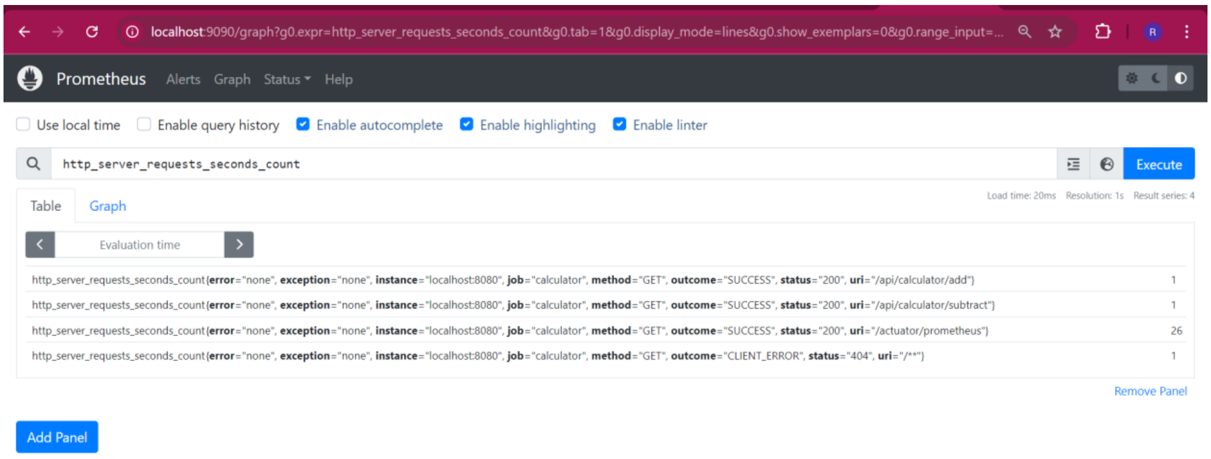
Query: `process_uptime_seconds`

Load time: 10ms Resolution: 14s Result series: 1

```
process_uptime_seconds{instance="localhost:8080", job="calculator"}    555.755
```



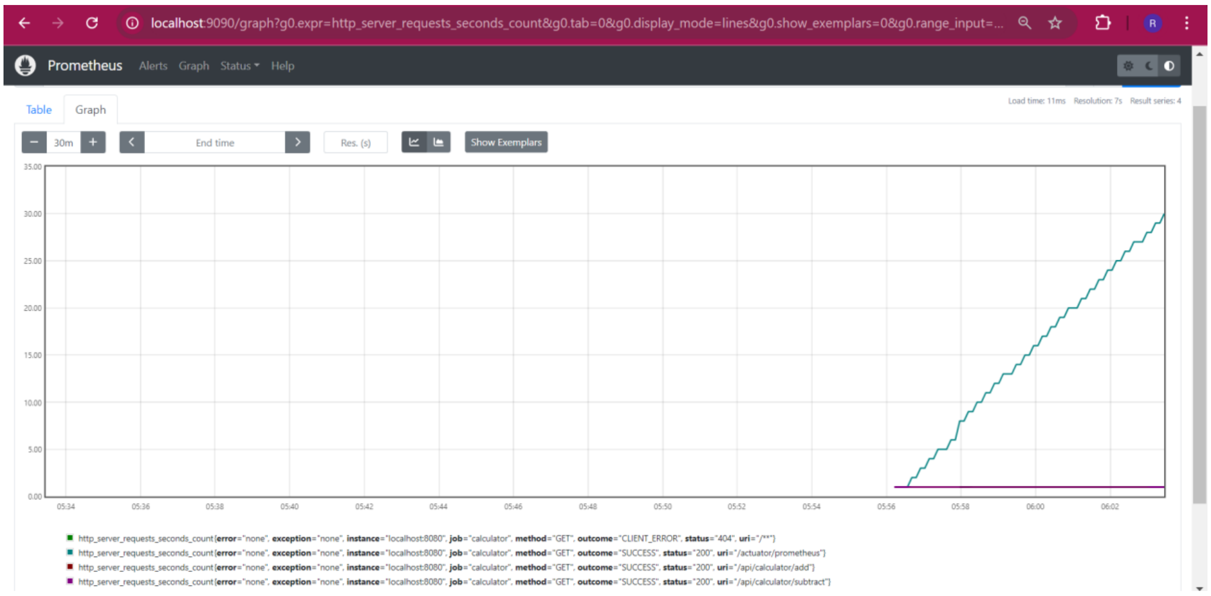
localhost:9090/graph?g0.expr=process_uptime_seconds&g0.tab=0&g0.display_mode=lines&g0.show_exemplars=0&g0.range_input=5m



```
http_server_requests_seconds_count
```

Load time: 20ms Resolution: 1s Result series: 4

```
http_server_requests_seconds_count{error="none", exception="none", instance="localhost:8080", job="calculator", method="GET", outcome="SUCCESS", status="200", uri="/api/calculator/add"}          1
http_server_requests_seconds_count{error="none", exception="none", instance="localhost:8080", job="calculator", method="GET", outcome="SUCCESS", status="200", uri="/api/calculator/subtract"}     1
http_server_requests_seconds_count{error="none", exception="none", instance="localhost:8080", job="calculator", method="GET", outcome="SUCCESS", status="200", uri="/actuator/prometheus"}         26
http_server_requests_seconds_count{error="none", exception="none", instance="localhost:8080", job="calculator", method="GET", outcome="CLIENT_ERROR", status="404", uri="/**"}                     1
```



Load time: 11ms Resolution: 7s Result series: 4

```
🟩 http_server_requests_seconds_count{error="none", exception="none", instance="localhost:8080", job="calculator", method="GET", outcome="CLIENT_ERROR", status="404", uri="/**"}
🟦 http_server_requests_seconds_count{error="none", exception="none", instance="localhost:8080", job="calculator", method="GET", outcome="SUCCESS", status="200", uri="/actuator/prometheus"}
🟥 http_server_requests_seconds_count{error="none", exception="none", instance="localhost:8080", job="calculator", method="GET", outcome="SUCCESS", status="200", uri="/api/calculator/add"}
🟪 http_server_requests_seconds_count{error="none", exception="none", instance="localhost:8080", job="calculator", method="GET", outcome="SUCCESS", status="200", uri="/api/calculator/subtract"}
```

CONCLUSION

The Calculator Application project demonstrates the effective use of Spring Boot for implementing a robust and efficient arithmetic calculator with integrated performance monitoring. By leveraging Micrometer for metrics collection and Prometheus for monitoring, the application provides real-time insights into the performance of arithmetic operations such as addition, subtraction, multiplication, and division.

The integration of Micrometer's @Timed annotations allows the tracking of operation durations, while counters provide data on the frequency of each operation. This setup facilitates detailed performance analysis and helps in identifying potential bottlenecks or issues in real-time. The modular design of the application includes a dedicated service for random number generation and a separate service for arithmetic operations, promoting clean separation of concerns and maintainability. The CalculatorController handles HTTP requests and utilizes Micrometer to capture metrics, ensuring that the application is well-instrumented for performance monitoring.

In summary, the Calculator Application project highlights the synergy between Spring Boot's powerful backend capabilities and Micrometer's metrics collection. This combination creates a scalable and insightful solution for performing and monitoring basic arithmetic operations, laying the groundwork for further enhancements or integration with additional monitoring and analysis tools.