Logging and Metrics

Introduction

Logging and metrics are essential aspects of modern software development and system administration. They play a crucial role in understanding the behavior and performance of applications and infrastructure, enabling developers and operators to monitor, troubleshoot, and optimize their systems effectively.

A. Logging

Logging is the practice of recording events, actions, and information in a system or application. These records, known as logs, are created for monitoring, troubleshooting, and auditing. Logging is a crucial aspect of software development and system management. It involves capturing various types of data, such as errors, warnings, informational messages, and performance metrics, to gain insights into the behaviour of an application or system.

Key points about logging:

1. **Log Levels**: Logs are categorized into different levels, such as DEBUG, INFO, WARN, and ERROR. Each level corresponds to a specific type of information, with ERROR being the most critical and DEBUG providing detailed debugging information.

1. **Log Messages**: Log messages contain relevant information about the events or conditions being logged. These messages help developers and system administrators understand what happens in an application.

2. **Log Frameworks:** Logging is typically implemented using logging frameworks or libraries. Java, for example, has popular logging frameworks like Logback, Log4j2, and SLF4J, which provide tools for generating and managing log data.

3. **Log Patterns:** Log frameworks allow developers to specify the format of log messages, including timestamps, log levels, class names, and custom text. Log patterns help in standardizing log output.

B. Logging Levels and Hierarchy:

In software development and system monitoring, different logging levels categorize log messages based on their significance and severity. These levels help developers and administrators understand the nature of logged events and determine how to respond to them. Here are some standard logging levels, from the most severe to the least severe:

1. **ERROR**: The ERROR level is the most critical and severe. It indicates that a significant issue or error has occurred in the application or system. These messages often require immediate attention to prevent adverse consequences. For example, an ERROR log might indicate a critical application crash or a major security breach.

2. **WARN** (Warning): The WARN level logs potentially problematic situations or events that are not critical but still require attention. These messages serve as early warnings about issues that could lead to errors or failures if not addressed. An example of a WARN log might be a system resource nearing its capacity limit.

3. **INFO** (Informational): The INFO level logs general information about the application's operation. These messages provide insight into the application's normal execution and help understand its behaviour. INFO logs can include information about successful operations, startup messages, and other relevant details.

4. **DEBUG**: DEBUG messages are used for detailed debugging information. They provide extensive insights into the application's inner workings, including variable values and program flow. These logs are helpful during development and debugging to identify and resolve issues.

5. **TRACE**: The TRACE level is the most detailed and lengthy. It is often used for extremely fine-grained logging, including method calls and loop iterations. TRACE logs are especially helpful for in-depth debugging and diagnosing complex issues.

Each logging level has a specific purpose in providing information about the application's behaviour. Developers and administrators can configure the logging system to capture log messages at various levels, depending on the desired level of detail and the urgency of response required. Properly utilizing these logging levels can aid in troubleshooting, error detection, and system monitoring.

Let's consider a sample scenario in a Spring Boot application where we want to log different events using different log levels.

```
import org.slf4j.Logger;
```

```
import org.slf4j.LoggerFactory;

public class SampleClass {

    private static final Logger logger =
LoggerFactory.getLogger(SampleClass.class);

    public void performActions() {
        logger.debug("This is a debug message.");
        logger.info("This is an info message.");
        logger.warn("This is a warning message.");
        logger.error("This is an error message.");
    }
}
```

You can use a logger in this way wherever required in your application to generate meaningful log statements.

Key points to remember:

1. You can define a log level for a particular file like this. (By default logging level is INFO)

```
logging:
  level:
    root: WARN # Set the default log level to WARN for all loggers.
    com.example.packageName: DEBUG # Set the log level to DEBUG for a
specific package or class.
```

2. After defining the log level for a particular package, only log levels with severity above them can be logged and not below. For example, defining log level as WARN for a package only loggers with ERROR would be printed.

C. Color Coding and Saving a Log File

Colour Coding in Logging:

Colour coding in logging is a technique where log messages are displayed with different text colors to distinguish log levels or categories visually. This practice enhances log readability and helps quickly identify the severity of log messages. Here are some common color coding conventions for log levels:

- INFO (Informational): Typically displayed in white or green text.
- DEBUG (Debugging): Often shown in blue or cyan.
- WARN (Warning): Usually displayed in yellow or orange.

- ERROR (Error): Shown in red.
- TRACE (Trace Information): Sometimes shown in gray or light blue.

To enable colored log statements in the console add this configuration to your '*application.yml*'

```
spring:
  output:
    ansi:
      enabled: ALWAYS
```

Saving Log Files:

Saving log files is a fundamental aspect of effective logging. Log files capture and store log messages for later analysis, auditing, and troubleshooting. Here are key points about saving log files:

- **Log Rotation**: To prevent log files from becoming too large and consuming excessive disk space, log rotation is implemented. This involves creating new log files periodically (e.g., daily or based on file size) and archiving or deleting old ones.
- **Log Retention Policies**: Establishing log retention policies is essential. These policies define how long log files should be retained. Some logs may need to be kept for a few days, while others should be archived for compliance or long-term analysis.
- **Log File Location:** Log files are typically stored in a dedicated directory or folder. It's important to define a clear and organized directory structure for log storage.

Sample yml config to save a log file:

```
logging:
  file:
    name: application.log
```

Note- files should be named with .log extension.
Effective log file management ensures that log data is preserved, secure, and readily available when needed, which is critical for maintaining the reliability and security of software applications.

D. Metrics

Metrics, in the context of software and system monitoring, are quantitative measurements used to track various aspects of application and system behavior. Metrics provide valuable data for assessing performance, identifying issues, and making informed decisions.

Types of Metrics:

- **Application Metrics**: Application-level metrics are specific to your Spring Boot application. They include information like the number of requests, response times, error rates, and more. Application metrics help you monitor the performance and behavior of your application in real time.

- **System Metrics**: System-level metrics pertain to the underlying infrastructure, such as the server or container where your application is running. These metrics include CPU usage, memory usage, disk I/O, and network traffic. System metrics help in understanding and optimizing resource utilization.

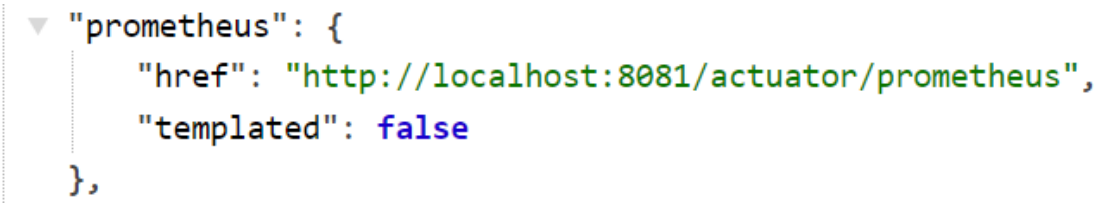
- **Business Metrics:** Business metrics are custom metrics that reflect specific key performance indicators (KPIs) for your application. They could be related to user engagement, sales, or any other business-specific goals. Business metrics provide insights into the impact of your application on your organization's objectives.

E. Micrometer - Metric Collection Library

Micrometer is a popular Java library for collecting application metrics. It provides tools for collecting, aggregating, and exporting metrics. They act as bridges between applications and monitoring systems, making it easier to track and analyze data. Micrometer supports multiple monitoring backends, including Prometheus, Grafana, and more.

Including this dependency in your application's pom.xml file will expose *"/prometheus"* endpoint below *"/actuator"* endpoint.

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
    <scope>runtime</scope>
</dependency>
```



F. Docker

Docker is a containerization technology that allows you to package applications and their dependencies into lightweight, portable containers. These containers can run consistently across different environments, from development to production. Docker containers offer process and file system isolation. They are highly portable, ensuring that applications run the same way on any system with Docker installed.

Creating images: Once installed, we can use docker to create images for Prometheus and Grafana and can run them locally without installing the software.

1. Install the Docker desktop on your local machine.
2. Create a file named ‘docker-compose.yml’. The yml code below sets up two services, Prometheus and Grafana, using their respective Docker images.

```
version: '3.7'

services:
  prometheus:
    image: prom/prometheus:v2.44.0
    container_name: prometheus
    ports:
      - "9090:9090"
    volumes:
      - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml

  grafana:
    image: grafana/grafana:9.5.2
    container_name: grafana
    ports:
      - "3000:3000"
    restart: unless-stopped
    volumes:
      -
./grafana/provisioning/datasources:/etc/grafana/provisioning/datasources
```

3. Open the command prompt or integrated terminal for the path in which docker-compose.yml is present and execute the command ‘docker compose up’.

G. Monitoring and Visualization tools

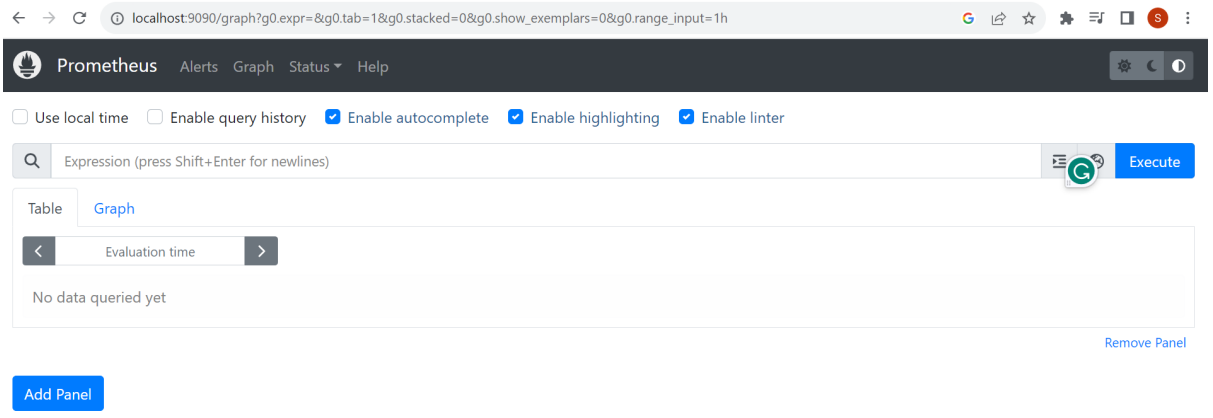
Prometheus

Prometheus is an open-source monitoring and alerting system designed for reliability, scalability, and real-time monitoring of various services and applications. It was originally developed by SoundCloud and is now part of the Cloud Native Computing Foundation (CNCF).

Key Features:

- **Time-Series Database**: Prometheus stores all collected data as time series, making it well-suited for time-based monitoring and analysis.
- **Data Scraping**: Prometheus scrapes metrics from instrumented applications and services at regular intervals.
- **Alerting**: Prometheus supports alerting based on user-defined rules, allowing you to set up notifications for critical events.
- **Highly Reliable:** It is designed to be highly available and resistant to failure, ensuring minimal data loss.

Prometheus dashboard:



Grafana

Grafana is an open-source platform used for monitoring, visualization, and observability of data from various sources. It is designed to provide a unified view of data from multiple systems, making it a valuable tool for monitoring and troubleshooting.

Key Features:

- Data Visualization: Grafana allows users to create interactive and customizable dashboards to visualize data.
- Data Sources: It supports a wide range of data sources, including Prometheus, InfluxDB etc.
- Alerting: Grafana can set up alerts and notifications based on defined rules and thresholds.
- Plugins and Extensibility: The platform supports plugins and extensions to add additional data sources, panels, and functionalities.
- User Collaboration: Teams can collaborate by sharing dashboards, creating annotations, and discussing data insights.
- Data Sources: Grafana can connect to various data sources, including time-series databases like Prometheus and InfluxDB, log aggregators like Elasticsearch, and relational databases. Users can create unified dashboards that combine data from different sources for comprehensive monitoring.
- Data Visualization: Grafana provides a wide range of visualization options, including line charts, bar charts, heatmaps, and more. It supports the creation of panels, which can display data from a selected data source. Users can customize visualization settings, such as colors, labels, and time intervals.
- User-Friendly Dashboards: Grafana's interface is user-friendly and provides an interactive experience for exploring and analyzing data. Dashboards are easy to create, modify, and share with team members.

Grafana dashboard Sample

H. Conclusion

In summary, effective logging, and comprehensive metrics collection are essential for maintaining application health, diagnosing issues, optimizing performance, and aligning software systems with organizational objectives. Additionally, tools like Prometheus and Grafana simplify the process of monitoring, analyzing, and visualizing data, enabling better decision-making and proactive issue resolution.

Instructor Codes

- Social Media Application

References

1. Logging: Official Documentation
2. Logging in Springboot
3. Metrics: Official Documentation
4. Micrometer
5. Prometheus from Micrometer
6. Docker: Documentation
7. Prometheus
8. Prometheus with Springboot
9. Grafana: Documentation
10. Prometheus vs Grafana