Chapter 7: MicroProfile Metrics

Introduction

This chapter provides a comprehensive and detailed overview of MicroProfile Metrics, a widely used specification for monitoring microservices. You will gain an understanding of the various types of metrics and how you can use them to monitor microservices effectively. Additionally, this chapter covers the standard metrics provided by MicroProfile and how you can leverage them to monitor various aspects of microservices.

Furthermore, this chapter discusses the process of instrumenting microservices, which involves adding code to the application to collect metrics. You will learn how to expose endpoints to access metric data and interpret the data generated by these metrics.

This chapter also highlights the importance of integrating monitoring solutions with MicroProfile Metrics. You will learn how to incorporate monitoring solutions and choose the right monitoring solution for your needs.

By the end of this chapter, you will have a deep understanding of MicroProfile Metrics and the various techniques for monitoring microservices. This chapter will equip you with the knowledge and skills to effectively monitor your microservices and ensure they perform optimally.

Topics to be covered:

- Introduction to MicroProfile Metrics
- Need for Metrics in Microservices
- Types of Metrics
- MicroProfile Metrics Dependency
- Metrics Annotations
- Using Metadata with Metrics
- Categories of Metrics
- Metric Registry
- Instrumenting Microservices with Metrics
- Creating Custom Metrics

Introduction to MicroProfile Metrics

It is essential to monitor its status regularly to ensure smooth microservice operations. You can monitor a microservice using two different techniques: Metrics and health checking. Health checks provide information on the health status of a

service, such as whether it is up and running, while Metrics offer more detailed information on its performance, such as response times, throughput, and error rates. In the previous chapter, we discussed health checks and their importance. This chapter will cover the MicroProfile Metrics specification, which provides a standardized way of collecting and exposing performance data for Java microservices.

MicroProfile Metrics is a specification for developers who want to measure their applications' performance more thoroughly. It provides a set of annotations and APIs to track various metrics related to the application's health and performance. For instance, developers can use these APIs to track metrics such as request counts, the response times, and the payload sizes, essential for real time performance monitoring and troubleshooting.

This specification defines a standardized format for exposing these metrics, which other tools and frameworks can easily collect and track. By using this specification, developers can monitor the performance of their applications in real time and identify any issues that may impact the user experience.

Moreover, this specification defines a set of standard metrics that we can expose in various formats, such as JMX (Java Management Extensions), JSON (JavaScript Object Notation), or Prometheus. Developers can choose the format that works best for their needs. With the help of this tool, developers can optimize their applications for better performance, ensuring that they meet the requirements of their consumers while delivering a seamless experience.

JMX is a comprehensive platform that provides developers the tools to monitor and manage Java applications efficiently, however its use is more common in traditional, monolithic architectures. JMX can be complex to handle remotely and does not integrate well outside JVM environments. MicroProfile Metrics provides support for more modern alternatives like JSON and Prometheus, facilitating metrics exposure through REST endpoints, which can be more scalable and flexible in cloud-native environments. This makes it more accessible and compatible in environments with multiple programming languages and technologies.

JSON allows developers to represent a wide range of data types, including strings, numbers, arrays, and objects. It is also widely supported across different programming languages and platforms, making it ideal for transmitting data between systems and platforms.

Prometheus is a powerful tool designed to monitor and collect metrics from your services. It provides a highly efficient time-series database system that securely stores your data for long-term analysis. With Prometheus, you can easily visualize and gain insights into your system's performance, allowing you to make informed decisions and optimize your services for better efficiency and reliability.

Need for Metrics in Microservices

Metrics, enables developers and operators to monitor and measure the behavior of microservices at runtime. This observability is crucial for:

- **Performance Tuning:** Identifying bottlenecks and optimizing resource utilization to ensure services are running efficiently.
- **Scalability:** Making informed decisions on when to scale services up or down based on real-time data on load and performance.
- **Troubleshooting:** Quickly pinpointing issues by analyzing trends in performance metrics, leading to reduced downtime.
- **Service Health Monitoring:** Complementing the MicroProfile Health checks by providing deeper insights into the internal state of a service, beyond simple up/down statuses.

Types of Metrics

MicroProfile Metrics offers a range of customizable metrics that can be used to measure and monitor microservices' performance. It allows developing microservices that are observable, manageable, and which provide insights into their behavior.

The MicroProfile Metrics specification includes four different types of metrics that serve specific monitoring purposes: `Counter`, `Gauge`, `Histogram`, and `Timer`. Each of these types offers unique insights into different aspects of application behavior and performance. Below is the breakdown of available metric types:

1. **Counter:** A Counter is a simple metric type that represents a single numerical value that can only increase or be reset to zero. It's typically used to count occurrences of certain events, such as the number of requests processed, items created, or tasks completed. Since counters can only go up, they are resettable to start counting from zero again, which is particularly useful for tracking totals over a specific interval.

2. **Gauge:** It is a metric type that measures an instantaneous value of something , which can arbitrarily go up or down. It’s used to capture the value of a metric at a particular point in time like the size of a queue, memory usage, or current number of active user sessions. Gauges are typically used for values that fluctuate over time and provide a current "gauge" of what's happening.

3. **Histograms**: They provide a distribution of values for a given metric, which are useful for identifying performance outliers. It measures the frequency of values in different ranges (or "buckets") and is useful for tracking the distribution of values, such as response times or data sizes. Histograms can give insights into the variability, average, median, percentiles, and trends of the measured data over time.

4. **Timer:** It is a specialized metric type that aggregates timing durations and provides count, total time, mean, minimum, and maximum duration, and it usually reports the duration distribution as well. It effectively combines a Histogram and a Meter, measuring the rate of events and the time duration of each event. Timers are invaluable for tracking the duration of certain activities

or operations within your application, such as processing time or method execution time.

By leveraging these metrics, developers and operators can gain a deeper understanding of how their microservices are performing. They can use this information to identify areas where improvements can be made and optimize their microservices' performance.

MicroProfile Metrics Dependency

If you're using Maven, add the following dependency to your `pom.xml` file located in the root folder of your project:

```
<dependency>
    <groupId>org.eclipse.microprofile.metrics</groupId>
    <artifactId>microprofile-metrics-api</artifactId>
    <version>5.1.1</version>
</dependency>
```

For Gradle, add the corresponding dependency to your `build.gradle` file located within the root folder of your project:

```
dependencies {
    providedCompile
'org.eclipse.microprofile.metrics:microprofile-metrics-api:5.1.1'
}
```

Metrics Annotations

MicroProfile Metrics defines a set of annotations to be used for exposing metrics. These annotations can be used on classes, methods, or fields. Table 7-1 shows the list of Metrics Annotation along with their descriptions.

Annotations	Descriptions
@Timed	It times how long a method takes to execute and exposes this information as a metric.
@Counted	It tracks how many times a method is invoked and exposes this information as a metric.
@Gauge	It allows you to expose a custom metric that can be any value. It is useful for exposing application-specific metrics.

Besides annotations, MicroProfile Metrics also defines a set of programmatic APIs for working with metrics. These APIs can be used to register custom metrics or access existing metrics.

Using Metadata for Metrics

In MicroProfile Metrics, metadata plays a crucial role in defining, understanding, and managing the various metrics that applications expose. Metadata essentially provides descriptive information about a metric, such as its name, description, type, unit of measurement, and additional tags or labels. This information not only aids in the identification and categorization of metrics but also enhances their interpretability, making it easier for developers, operators, and monitoring tools to understand what each metric represents and how it should be used.

Components of Metadata

- **Name:** A unique identifier for the metric. Naming conventions are important for clarity and consistency.
- **Description:** A human-readable description of what the metric measures and any additional context.
- **Type:** Specifies the type of metric (e.g., Counter, Gauge, Timer, etc.), which determines how its data should be interpreted.
- **Unit of Measurement:** Indicates the unit in which the metric value is reported (e.g., milliseconds, bytes, requests per second), crucial for understanding the scale and magnitude of the metric.
- **Tags/Labels:** Key-value pairs that provide additional dimensions to a metric, such as the module or component it relates to, enabling more granular analysis and filtering.

Example

Consider a metric that tracks the number of user login attempts in a web application. The metadata for this metric might look like this:

- Name: `login_attempts`
- Description: "Counts the number of user login attempts, including successes and failures."
- Type: `Counter`
- Unit: `attempts`
- Tags: `{ action="login", outcome="success|failure" }`

This metadata tells us that `login_attempts` is a counter metric, meaning it will increment over time, tracking login attempts. The description provides context on what is being counted, and the tags allow for differentiation between successful and failed attempts.

Purpose and Importance of Metadata

- **Identification:** Metadata helps uniquely identify a metric within the application or across different systems. The name and tags associated with a metric can precisely pinpoint its source and purpose.
- **Documentation:** Through descriptive texts and labels, metadata serves as documentation for metrics, explaining their meaning, usage, and any relevant context that users might need to interpret the data correctly.
- **Standardization:** By defining expected metadata for metrics, MicroProfile Metrics promotes standardization across applications and tools, facilitating easier integration with monitoring and analysis tools.
- **Categorization and Filtering:** Metadata allows metrics to be categorized and filtered based on tags or other attributes. This is particularly useful in complex systems with numerous metrics, enabling users to focus on the most relevant data points.
- **Configuration:** Some metadata fields can influence how a metric is collected, stored, or displayed by monitoring systems, offering a degree of control over the metrics pipeline.

Categories of Metrics

In MicroProfile Metrics, metrics are organized into three distinct scopes: Base, Vendor, and Application. This categorization is designed to clearly separate metrics by their origin and relevance, making it easier for developers and operators to monitor and manage the performance of their microservices. Each scope serves a specific purpose and contains a different set of metrics:

- **Base Metrics** are common to all applications, such as the number of CPUs or the amount of free memory. These metrics provide essential information about the underlying Java Virtual Machine (JVM) and the core libraries that are common across all MicroProfile applications. Base metrics typically include JVM-specific metrics such as memory usage, CPU load, thread counts, and garbage collection statistics. The intention behind base metrics is to offer a consistent set of low-level metrics that are universally applicable and useful for monitoring the health and performance of the JVM itself, which is the foundation upon which all MicroProfile applications run.

  Base metrics are exposed under the path `/metrics?scope=base`.

- **Application Metrics** are specific to an application, they are defined by the developers of the MicroProfile applications themselves. These are custom metrics that are specific to the business logic or operational aspects of the application. Developers use annotations or programmatic APIs to create and register these metrics, tailoring them to monitor the performance and behavior of their application's unique functionalities. Application metrics enable developers to gain insights into the runtime characteristics of their application, such as the number of transactions processed, response times for specific endpoints, or the rate of specific business events.

Application metrics are exposed under the path `/metrics?scope=application`.

- **Vendor Metrics** are specific to a particular vendor or technology. These metrics provide insights into the performance and behavior of the runtime's internal components and extensions. Since different MicroProfile implementations may offer additional features or optimize certain areas differently, vendor metrics can vary widely between runtimes. They allow runtime vendors to expose unique metrics that are relevant to their implementation, offering users the ability to monitor vendor-specific aspects of their applications.

  Application metrics are exposed under the path `/metrics?scope=vendor`.

Besides the standard metrics above, MicroProfile Metrics also supports custom metrics. You can use custom metrics to track application-specific information not covered by the standard metrics.

Metric Registry

The **MetricRegistry** component acts as a container for storing and managing metrics within an application. It provides a structured way to collect, organize, and access various types of metrics (e.g., counters, gauges, histograms, timers, and metered metrics) for monitoring the behavior and performance of applications. It offers a centralized repository where metrics can be registered, updated, and retrieved. This allows applications to consistently monitor critical operational and performance statistics.

Types of Metric Registries

MicroProfile Metrics defines several types of registries, categorized by their scope:

- Application Scope (`MetricRegistry.Type.APPLICATION`): Contains custom metrics that are specific to the application. These are typically the metrics that developers explicitly create and register to monitor application-specific behaviors.
- Base Scope (`MetricRegistry.Type.BASE`): Contains metrics that are fundamental and common across all MicroProfile applications. These metrics provide basic information about the underlying JVM and application server.
- Vendor Scope (`MetricRegistry.Type.VENDOR`): Contains metrics that are specific to the implementation of the MicroProfile platform being used. These metrics offer insights into vendor-specific features and optimizations.

Instrumenting Microservices with MicroProfile Metrics

Instrumenting microservices with MicroProfile Metrics enables developers to gain detailed insights into their application's operational health and performance. This level of observability is essential for maintaining scalable and resilient microservice architectures in dynamic environments.

Tracking response time using @Timed

MicroProfile Metrics also allows you to track a method's response time as a timed metric. The code example below shows how to use the `@Timed` annotation to track the response time.

```
import org.eclipse.microprofile.metrics.annotation.Timed;
// ...

public class ProductResource {

    // ...
    // Expose the response time as a timer metric
    @Timed(name = "productLookupTime",
            tags = {"method=getProduct"},
            absolute = true,
            description = "Time needed to lookup for a products")
    public Product getProduct(@PathParam("id") Long productId) {
        return productService.getProduct(productId);
    }

    // ...
```

It will expose a metric called `productLookupTime`, which will track the amount of time spent in the `getProduct()` method in milliseconds.

You can visit the following URL `https://localhost:<port>/metrics?scope=application` (Replace <port> with the actual port where the server is running) to see the Response time of this method as below:

```
...
# HELP productLookupTime_seconds_max Time needed to lookup for a
products
# TYPE productLookupTime_seconds_max gauge
productLookupTime_seconds_max{method="getProduct",mp_scope="applic
ation",} 0.002270643
...
```

Tracking number of invocations

MicroProfile Metrics also allows you to track the number of invocations of a method as a counter metric. The code example below shows how to use the `@Counted` annotation to track the invocation count.

```
import org.eclipse.microprofile.metrics.Metrics;
import org.eclipse.microprofile.metrics.MetricType;

public class ProductResource {

    // Expose the invocation count as a counter metric
    @Counted(name = "productAccessCount",
            absolute = true,
            description = "Number of times the list of products is requested")
    public Response getProducts() {
       // Method implementation
       // ....
    }
}
```

In the example above, the `@Counted` annotation tells MicroProfile Metrics to track the number of invocations of the `getProducts()` method and expose this metric as a counter. The name, and description of the metric can also be specified.

You can visit the following URL `https://localhost:<port>/metrics?scope=application` (Replace <port> with the actual port where the server is running) to see the number of times this method is called as below:

```
...
# HELP productAccessCount_total Number of times the list of
products is requested
# TYPE productAccessCount_total counter
productAccessCount_total{mp_scope="application",} 3.0
...
```

Creating a Custom Metric

Creating a custom metric to track the number of products in a catalog within a `ProductService` microservice involves using the MicroProfile Metrics API. This custom metric can be implemented as a gauge, which measures an instantaneous value (in this case, the current number of products in the catalog).

```
import org.eclipse.microprofile.metrics.annotation.Gauge;
...

@Path("/products")
@ApplicationScoped
```

```
public class ProductResource {
      // ...


   @GET
   @Path("/count")
   @Produces(MediaType.APPLICATION_JSON)
   @Gauge(name = "productCatalogSize",
unit = "none",
description = "Current number of products in the catalog")
   public long getProductCount() {
       return productCatalogSize;
   }
```

The gauge metric `productCatalogSize` can be accessed through the following endpoint:

```
/metrics?name=io_microprofile_tutorial_store_product_resource_ProductResource_pr
oductCatalogSize
```

This custom metric implementation provides a real-time insight into the size of your product catalog, which can be invaluable for monitoring the scale of your service's data and understanding its behavior over time.

Summary

This Chapter delved into the intricacies of MicroProfile Metrics, illuminating its role as a pivotal specification for efficiently monitoring microservices. Now you are equipped with a thorough understanding of diverse metric types and their application for monitoring microservice performance. This chapter highlighted the need of regular microservice monitoring via metrics and health checks, emphasizing metrics for detailed performance insights such as response times and throughput. Through practical examples, this chapter showcases how to instrument microservices with MicroProfile Metrics, leveraging standard metrics, and creating custom metrics to monitor microservices comprehensively.