The Prometheus Java Client and Micrometer

Author	Fabian Stäber
Date	2021-02-26
Formatting	Presentation of the current status is in black. Opinion and proposals for future development are in green.
Audience	Developers who are familiar with the Prometheus Java client library but not necessarily with Micrometer or the Java ecosystem.
Terminology	• Prometheus Java client: https://github.com/prometheus/client_java • Micrometer's Prometheus meter registry: https://github.com/micrometer-metrics/micrometer/tree/master/implementations/micrometer-registry-prometheus

Abstract

This document describes the current status of the Prometheus Java client, its relation to Micrometer, and plans for future development.

Micrometer Overview

Micrometer is a vendor-agnostic metrics library written in Java that was developed by the Spring project as part of Spring Boot 2 under the Apache 2.0 license. It ships with Spring boot, and has been adapted by other frameworks like Micronaut or RedHat's Quarkus.
Micrometer is popular in Java frameworks because it allows them to implement built-in metrics against a stable API without binding the framework to a specific monitoring system. Users can configure Micrometer to export Prometheus metrics, but it also provides meter registries for more than a dozen other vendors[1].
Micrometer's Api defines interfaces for Meters. There are different types of Meters:

- Counter meters, corresponding to Prometheus counters.
- Gauge meters, corresponding to Prometheus gauges.
- Timer meters for measuring time intervals, corresponding to Prometheus histograms or summaries.
- Distribution summary meters use the same internal data model as timers, but for things that don't represent time, e.g. request sizes.

[1] AppOptics, Atlas, Datadog, Dynatrace, Elastic, Ganglia, Graphite, Humio, Influx, Instana, JMX, KairosDB, New Relic, SignalFx, Stackdriver, StatsD, Wavefront.

Meters support tags which have the same semantics as Prometheus' dimensional metric data model.
Vendors provide implementations for these interfaces in a vendor-specific meter registry. For example, there is a `micrometer-registry-prometheus` module, maintained by the Micrometer project, implementing a meter registry for Prometheus metrics.
Vendor-specific meter registries differ a lot in their internal representation of the metrics. For example, the Prometheus meter registry represents counter meters as monotonously increasing Prometheus counters, while other vendors may decide to represent their counter meters as aggregated rate metrics.

Data Model

Comparison of the Data Model Implementations

Micrometer's Prometheus meter registry provides its own implementation of the metric types. It does not use `Counter`, `Gauge`, `Histogram`, and `Summary` from the Prometheus Java library. The following table shows how the original Prometheus data types are implemented in Micrometer's Prometheus meter registry.

Prometheus' Java Library	Micrometer's Prometheus Meter Registry
Counter	io.micrometer.prometheus.PrometheusCounter Very similar to the counter implementation in the Prometheus Java client, based on the same underlying `DoubleAdder`.
Gauge	io.micrometer.core.instrument.internal.DefaultGauge Implementation is not specific to the Prometheus meter registry, but used in meter registries from multiple vendors.
Histogram Summary (GaugeHistogram)	io.micrometer.prometheus.PrometheusTimer The underlying representation in Micrometer is an Hdr histogram (to be precise: Several Hdr histograms representing sliding time windows). The implementation of the Hdr histogram is not specific to the Prometheus meter registry, it's used in meter registries from multiple vendors. It is highly configurable how the data from the internal Hdr histograms are exported: ● Per default you always get some simple base metrics like min, max, count. ● Hdr histograms can be exported as Prometheus summaries, Micrometer has API for specifying the desired quantiles. ● Hdr histograms can be exported as Prometheus histograms with the original Hdr histogram buckets.

	This is useful when the user intends to use the `histogram_quantile()` function in PromQL, potentially after aggregating multiple histograms. ● The Hdr histograms support configurable custom bucket boundaries, which is useful if an SLO defines bucket boundaries that are not represented in the default Hdr histogram implementation. io.micrometer.prometheus.PrometheusDistributionSummary Same underlying Hdr histograms, but used for distributions that do not represent latencies, e.g. request sizes.
(Info) (StateSet)	counter (?)

Future Plans for the Data Model

As shown above, Micrometer does not use the `Counter`, `Gauge`, `Histogram`, and `Summary` implementations from the Prometheus Java client library. Here are some ideas on maintaining the data model of the Prometheus Java client library in the future:

- We want the Prometheus Java client library to remain usable without Micrometer. Thus we will keep maintaining the implementation of the data model in the Prometheus Java client library.
- The Prometheus Java client library recently introduced support for the new OpenMetrics types: `Info`, `StateSet`, and `GaugeHistogram`. We should aim at making these available via the Micrometer API. This will be a PR for Micrometer's Prometheus meter registry, not for the Prometheus Java client.
- If Björn's "sparse high-resolution histograms"[2] become accepted, we will need to provide a Java implementation, and this implementation will need to replace the current Hdr histograms in Micrometer's Prometheus registry. As Micrometer calculates quantiles from Hdr diagrams, replacing Hdr diagrams will require switching to Prometheus Summaries for quantiles in Micrometer's Prometheus registry as well. As a result, Micrometer's Prometheus registry will switch from using mostly its own internal data model to using the future Prometheus client library's model. There will be some discussion needed with the Micrometer project, because the user-facing API for configuring histograms in Micrometer does not match the planned configuration options for the "sparse high-resolution histograms".

[2] https://docs.google.com/document/d/1cLNv3aufPZb3fNfaJgdaRBZsInZKKIHo9E6HinJVbpM/edit

Exposition Format

Micrometer uses the Prometheus Java client library as a dependency, and calls `TextFormat.write004()` directly when scraped. The Prometheus Java client library will continue to provide and maintain the Prometheus and OpenMetrics exposition formats.

Built-In Metrics

Built-In Metrics Provided via Micrometer

Users may use Micrometer via direct API calls, which is good for monitoring custom business logic. However, in practice most Micrometer metrics are provided by Java frameworks and libraries out-of-the-box. Users can get very good monitoring coverage just by enabling the frameworks' built-in metrics without writing a single line of code. Micrometer provides a flexible `MeterBinder` abstraction that can be used by frameworks to provide metrics.
As these metrics are maintained by the frameworks and libraries, and not by the Micrometer project itself, it is hard to list all built-in Micrometer metrics. To give a glimpse, here's a list of Micrometer metrics provided by Spring boot, taken from the Spring boot documentation:

- VM metrics (various memory and buffer pools, garbage collection, threads utilization, number of classes loaded/unloaded)
- CPU metrics
- File descriptor metrics
- Kafka consumer, producer, and streams metrics
- Logging metrics: Log4j2, Logback
- Tomcat metrics
- Spring Integration metrics
- Spring MVC Metrics
- Spring WebFlux Metrics
- Jersey Server Metrics
- HTTP Client Metrics
- Cache Metrics (including Caffeine)
- DataSource Metrics
- Hibernate Metrics
- RabbitMQ Metrics
- Kafka Metrics

This list is incomplete, there are tons of other frameworks providing metrics via Micrometer.

Instrumentation Libraries provided by the Prometheus Java Client Project

Apart from the core Java client library, the Prometheus Java client Github repository contains several instrumentation libraries for popular Java libraries:

- Caffeine
  - Caffein is a cache implementation in Java
  - `simpleclient_caffeine` is an exporter for monitoring that cache
- Dropwizard metrics
  - Dropwizard is a Java metrics library developed as part of the Dropwizard framework
  - `simpleclient_dropwizard` is a mapper from Dropwizard metrics to Prometheus.
- Guava
  - Guava is a cache implementation in Java
  - `simpleclient_guava` is an exporter for monitoring that cache
- Hibernate
  - Hibernate is an ORM (used for accessing SQL databases from Java)
  - `simpleclient_hibernate` is an exporter for monitoring Hibernate
- Hotspot
  - Hotspot is the most widely used Java virtual machine.
  - `simpleclient_hotspot` is an exporter for monitoring Hotspot (garbage collection activity, etc.)
- Jetty
  - Jetty is an HTTP server and Servlet container
  - `simpleclient_jetty` is an exporter for monitoring Jetty statistics
- Log4j, etc.
  - Log4j, log4j2, and logback are logging libraries
  - `simpleclient_log4j`, etc. are exporters counting the number of errors, warnings, infos, etc.
- Spring Boot
  - Spring boot is the most popular Java framework
  - `simpleclient_spring_boot` is an exporter for Spring boot metrics. It was implemented in 2016 for Spring boot 1, which has reached its end of life.
- Spring Web
  - Spring Web is a sub-project within the Spring framework for writing HTTP controllers
  - `simpleclient_spring_web` provides an annotation to export execution times for HTTP controller calls. Like simpleclient_spring_boot, it was written before Spring boot 2 and Micrometer came out.

Future Maintenance of the Prometheus Java Instrumentation Libraries

Not sure whether we should maintain the instrumentation libraries provided by the Prometheus Java client project. We could consider deprecating them. The implementation of these libraries is mostly only a few lines of sample code, which could be converted into examples in Markdown

documentation. For example, a documentation on "How to monitor log messages in log4j" explaining the sample code might be more valuable than an actual log4j monitoring library with very limited functionality.
In cases where upstream projects provide Micrometer bindings, we should encourage users to use Micrometer for exporting Prometheus metrics. It is likely that upstream projects are better at providing meaningful metrics for their own projects than the Prometheus community.

Built-In Exporters

The Prometheus Java client project includes a number of built-in exporters for making metrics available to the Prometheus server or to Graphite:

- Graphite bridge: Pushes metrics collected with the Prometheus Java client library into Graphite.
- HTTP Server: Standalone HTTP server for exporting the metrics.
- Pushgateway: Library for pushing metrics into the Prometheus push gateway.
- Servlet: Exporter servlet to be deployed in a Servlet container.
- Vert.x: Handler for creating a metrics endpoint in the Vert.x Web framework.

The exporters should be kept and maintained, because without them users will not be able to get metrics out of the Prometheus Java client library without using Micrometer.

Other Future Topics

Exemplars

Exemplars have been defined in OpenMetrics as a way of linking metrics to example trace ids. Currently, exemplars are neither implemented in the Prometheus Java client library nor in Micrometer. It is desirable to add exemplar support.

Java Flight Recorder (JFR) Event Streaming

JFR event streaming is a fairly new feature introduced in Java 14. It might be worthwhile to explore how to monitor JFR events with Prometheus. Results should be done in form of a tutorial documentation, and not providing a ready to use instrumentation library.