Metrics for Eclipse MicroProfile

The MicroProfile community and its contributors

Version 5.0.0, November 04, 2022: Final

Table of Contents

Specification: Metrics for Eclipse MicroProfile

Version: 5.0.0

Status: Final

Release: November 04, 2022

Copyright

Eclipse Foundation Specification License

Disclaimers

Introduction

To ensure reliable operation of software it is necessary to monitor essential system parameters. This enhancement proposes the addition of well-known monitoring endpoints and metrics for each process adhering to the Eclipse MicroProfile standard.

This proposal does not talk about health checks. There is a separate specification for Health Checks.

In the previous release we mentioned our intent to investigate Micrometer. This has led to key changes in this specification, and the corresponding API, to allow for a variety of possible implementations. As examples, it should be possible to implement this specification using metrics libraries from Micrometer or Open Telemetry. The modifications to the API, since the previous release, have been made in consideration of maintaining backwards compatibility, as much as possible, but while removing parts of the API that limited the ability to plug in new implementations.

Motivation

Reliable service of a platform needs monitoring. There is already JMX as standard to expose metrics, but remote-JMX is not easy to deal with and especially does not fit well in a polyglot environment where other services are not running on the JVM. To enable monitoring in an easy fashion it is necessary that all MicroProfile implementations follow a certain standard with respect to (base) API path, data types involved, always available metrics and return codes used.

Difference to health checks

Health checks are primarily targeted at a quick yes/no response to the question "Is my application still running ok?". Modern systems that schedule the starting of applications (e.g. Kubernetes) use this information to restart the application if the answer is 'no'.

Metrics on the other hand can help to determine the health. Beyond this they serve to pinpoint issues, provide long term trend data for capacity planning and pro-active discovery of issues (e.g. disk usage growing without bounds). Metrics can also help those scheduling systems decide when to scale the application to run on more or fewer machines.

Architecture

This chapter describes the architectural overview of how metrics are setup, stored and exposed for consumption. This chapter also lists the various scopes of metrics.

See section Base Metrics for more information regarding spec-described metrics that vendors may provide.

See section Application Metrics Programming Model for more information regarding the application metrics programming model.

Metrics Setup

Metrics that are exposed need to be configured in the server. On top of the pure metrics, metadata needs to be provided.

The following three sets of sub-resource (scopes) are exposed.

- base: spec-described metrics that vendors may provide (optional)
- vendor: vendor specific metrics (optional)
- application: application-specific metrics (optional)

Scopes

Base metrics

Base metrics describe a set of optional metrics that MicroProfile-compliant servers may provide. Vendors may choose to implement all of the base metrics, some of them, or none of them. They are provided to ensure consistency between vendors that choose to implement them. One reason vendors may choose to use a different set of metrics is that the metrics libraries they are using in their implementation come with metrics with a similar purpose to the spec-described base metrics. If vendors choose to use different metrics, for example to measure values related to the JVM, they must do so using the *vendor* scope for those metrics. If vendors choose to use these base scope metrics, they can implement the set-up of the metrics in a vendor-specific way. The metrics can be hard coded into the server or read from a configuration file or supplied via the Java-API described in Application Metrics Programming Model. The Appendix shows a possible data format for such a configuration. The configuration and set up of the base scope is thus an implementation detail and is not expected to be portable across vendors.

Section Base Metrics lists the base metrics.

The implementation must tag base metrics with `mp_scope=base`.

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

The `base` scope is used for, and only for, any metrics that are defined in MicroProfile specifications. Metrics in the base scope, where implemented, are intended to be portable between different MicroProfile-compatible runtimes at the same version.

Application metrics

Application specific metrics can not be baked into the server as they are supposed to be provided by the application at runtime. Therefore a Java API is provided. Application specific metrics are supposed to be portable to other MicroProfile implementations. That means that an application written to this specification which exposes metrics, can expose the same metrics on a different compliant server without change.

Details of this Java API are described in Application Metrics Programming Model.

The implementation must automatically tag application metrics with `scope=application`, except where the metric already has a `scope` tag.

Metrics with application scope are exposed under `/metrics?scope=application`.

Applications may also define their own scope names by tagging metrics with another `scope` (other than `base`, `vendor`, `application`). Metrics with custom scopes are exposed under `/metrics?scope=scope_name` where scope_name is defined by the application. Scope names must match the regex `[a-zA-Z_][a-zA-Z0-9_]*`. If an illegal character is used, the implementation must throw an `IllegalArgumentException`.

Metrics with application-defined scopes are exposed under `/metrics?scope=scope_name`

Vendor specific Metrics

Vendors may choose to provide metrics relevant to their runtime.

Vendor specific metrics are exposed under `/metrics?scope=vendor`.

Examples for vendor specific data could be metrics like:

- OSGi statistics if the MicroProfile-enabled container internally runs on top of OSGi.
- Statistics of some internal caching modules
- Any other metrics that are generated by application frameworks, but not directly declared in application code, if these metrics are not based on any specification and therefore not expected to be portable between different runtimes that might support the same application framework.

Vendor specific metrics are not supposed to be portable between different implementations of MicroProfile servers, even if they are compliant with the same version of this specification.

Vendors are encouraged to use metric names consistent with the Open Telemetry Metrics Semantic Conventions where applicable.

Tags

Tags (also known as labels) play an important role in modern microservices and microservice scheduling systems (like e.g. Kubernetes). Application code can run on any node and can be re-scheduled to a different node at any time. Each container in such an environment gets its own ID; when the container is stopped and a new one started for the same image, it will get a different id. The classical mapping of host/node and application runtime on it, therefore no longer works.

Tags have taken over the role to, for example, identify an application (`app=myShop`), the tier inside the application (`tier=database` or `tier=app_server`) and also the node/container id. Metric value aggregation can then work over label queries (Give me the API hit count for `app=myShop && tier=app_server`).

In MicroProfile Metrics, tags add an additional dimension to metrics that share a common basis. For example, a metric named `carCount` can be further differentiated by the car type (sedan, SUV, coupe, and etc) and the colour (red, blue, white, black, and etc). Rather than incorporating this in the metric name, tags can be used to capture this information in separate metrics.

```
carCount{type=sedan,colour=red}
carCount{type=sedan,colour=blue}
carCount{type=suv,colour=red}
carCount{type=coupe,colour=blue}
```

For portability reasons, the key name for the tag must match the regex `[a-zA-Z_][a-zA-Z0-9_]*` (Ascii alphabet, numbers and underscore). If an illegal character is used, the implementation must throw an `IllegalArgumentException`. If a duplicate tag is used, the last occurrence of the tag is used.

The tags named `mp_scope` and `mp_app` are reserved. If an application attempts to create a metric with either of these tags, the implementation must throw an `IllegalArgumentException`.

The tag value may contain any UTF-8 encoded character.

The REST endpoints provided by MicroProfile Metrics have different reserved characters based on the format. The characters are only escaped as needed when exposed through the REST endpoints. See REST endpoints for more information on the reserved characters.

Tags can be supplied in two ways:

- At the level of a metric as described in Application Metrics Programming Model.
- At the application server level by using MicroProfile Config and setting a configuration property of the name `mp.metrics.tags`. The implementation MUST make sure that an implementation of MicroProfile Config version at least 2.0 is available at runtime. If it is supplied as an environment variable rather than system property, it can be named `MP_METRICS_TAGS` and will be picked up too.
    - Tag values set through `mp.metrics.tags` MUST escape equal symbols `=` and commas `,` with a backslash `\`

Set up global tags via environment variable

```
export MP_METRICS_TAGS=app=shop,tier=integration,special=deli\=ver\,y
```

Global tags and tags registered with the metric are included in the output returned from the REST API.

Global tags MUST NOT be added to the `MetricID` objects. Global tags must be included in list of tags when metrics are exported.

In application servers with multiple applications deployed, values of the reserved tag `mp_app` distinguish metrics from different applications and must not be used for any other purpose. For details, see section Usage of MicroProfile Metrics in application servers with multiple applications.

Metadata

Metadata can be specified for metrics in any scope. For base metrics, metadata must be provided by the implementation. Metadata is exposed by the REST handler.

While technically it is possible to expose metrics without (some) of the metadata, it helps tooling and also operators when correct metadata is provided, as this helps getting a context and an explanation of the metric.

The Metadata:

- name: The name of the metric.
- unit: a fixed set of string units
- description (optional): A human readable description of the metric.

Metadata must not change over the lifetime of a process (i.e. it is not allowed to return the units as seconds in one retrieval and as hours in a subsequent one). The reason behind it is that e.g. a monitoring agent on Kubernetes may read the metadata once it sees the new container and store it. It may not periodically re-query the process for the metadata.

In fact, metadata should not change during the life-time of the whole container image or an application, as all containers spawned from it will be "the same" and form part of an app, where it would be confusing in an overall view if the same metric has different metadata.

Metric Registry

The `MetricRegistry` stores the metrics and metadata information. There is one `MetricRegistry` instance for each of the predefined scopes listed in Scopes.

Metrics can be added to or retrieved from the registry either using the `@Metric` annotation (see Metrics Annotations) or using the `MetricRegistry` object directly.

A metric is uniquely identified by the `MetricRegistry` if the `MetricID` associated with the metric is unique. That is to say, there are no other metrics with the same combination of metric name and tags. However, all metrics of the same name must be of the same type and be identified by the same set of tag names otherwise an `IllegalArgumentException` will be thrown. This exception will be thrown during registration.

The metadata information is registered under a unique metric name and is immutable. All metrics of the same name must be registered with the same metadata information otherwise an "IllegalArgumentException" will be thrown. This exception will be thrown during registration.

MetricID

The MetricID consists of the metric's name and tags (if supplied). This is used by the MetricRegistry to uniquely identify a metric and its corresponding metadata.

The MetricID:

- name: The name of the metric.
- tags (optional): A list of Tag objects. See also Tags.

Reusing Metrics

For metrics declared using annotations, it is allowed to reference one metric by multiple annotations. The prerequisite for this is that all such annotations must carry the same metadata and tag names. If multiple annotations declare the same metric but contain different metadata or tag names, an IllegalArgumentException must be thrown during startup.

Reusability does not apply to gauges though. The implementation must throw an `IllegalArgumentException` during startup if it detects multiple `@Gauge` annotations referring to the same gauge (with the same `MetricID`).

Example of reused counters

```
@Counted(name = "countMe", absolute = true, tags={"tag1=value1"})
public void countMeA() { }

@Counted(name = "countMe", absolute = true, tags={"tag1=value1"})
public void countMeB() { }
```

In the above examples both `countMeA()` and `countMeB()` will share a single Counter with registered name `countMe` and the same tags in application scope.

Metrics and CDI scopes

Depending on CDI bean scope, there may be multiple instances of the CDI bean created over the lifecycle of an application. In these cases, where multiple bean instances exist, only one instance of the corresponding metric will be created (per annotated method), and updates to that metric will be combined from all related invocations regardless of the bean instance where the invocation happens. For example, calls to a method annotated with `@Counted` will increase the value of the same counter no matter which bean instance is the one where the counted method is being invoked.

The only exception from this are gauges, which don't support multiple instances of the underlying bean to be created, because in that case it would not be clear which instance should be used for obtaining the gauge value. For this reason, gauges should only be used with beans that create only

one instance, in CDI terms this means `@ApplicationScoped` and `@Singleton` beans. The implementation may employ validation checks that throw an error eagerly when it is detected that there is a `@Gauge` on a bean that will probably have multiple instances.

Exposing metrics via REST API

Data is exposed via REST over HTTP under the `/metrics` base path in different data formats for `GET` requests:

- OpenMetrics exposition format - used when the HTTP Accept header best matches `application/openmetrics-text; version=1.0.0`. Support for this format by implementations is optional.
- Prometheus text-based exposition format - used when the HTTP Accept header best matches `text/plain; version=0.0.4`. This format is also returned when no media type is requested (i.e. no Accept header is provided in the request)

Implementations and/or future versions of this specification may allow for more export formats that are triggered by their specific media type. The Prometheus text-based exposition format will stay as fall-back.

Formats are detailed below.

Data access must honour the HTTP response codes, especially

- 200 for successful retrieval of an object
- 204 when retrieving a subtree that would exist, but has no content. E.g. when the application-specific subtree has no application specific metrics defined.
- 404 if a directly-addressed item does not exist. This may be a non-existing sub-tree or non-existing object
- 406 if the HTTP Accept Header in the request cannot be handled by the server.
- 500 to indicate that a request failed due to "bad health". The body SHOULD contain details if possible { "details": <text> }

The API MUST NOT return a 500 Internal Server Error code to represent a non-existing resource.

Table 1. Supported REST endpoints

Endpoint	Request Type	Supported Formats	Description
`/metrics`	GET	Prometheus, OpenMetrics	Returns all registered metrics
`/metrics?scope=<scope_name>`	GET	Prometheus, OpenMetrics	Returns metrics registered for the respective scope. Scopes are listed in Metrics Setup
`/metrics?scope=<scope_name>&name=<metric_name>`	GET	Prometheus, OpenMetrics	Returns metrics that match the metric name for the respective scope

Usage of MicroProfile Metrics in application servers with multiple applications

Even though multi-app servers are generally outside the scope of MicroProfile, this section describes recommendations how such application servers should behave if they want to support MicroProfile Metrics.

Metrics from all applications and scopes should be available under a single REST endpoint ending with `/metrics` similarly as in case of single-application deployments (microservices).

To help distinguish between metrics pertaining to each deployed application, a tag named `mp_app` should be added to each metric.

The value of the `mp_app` tag should be passed by the application server to the application via a MicroProfile Config property named `mp.metrics.appName`. It should be possible to override this value by bundling the file `META-INF/microprofile-config.properties` within the application archive and setting a custom value for the property `mp.metrics.appName` inside it.

It is allowed for application servers to choose to not add the `mp_app` tag at all. Implementations may differ in how they handle cases where metrics are registered with the same name from two or more applications running in the same server. This behavior is not expected to be portable across vendors.

REST endpoints

This section describes the REST-API, that monitoring agents would use to retrieve the collected metrics. (Java-) methods mentioned refer to the respective Objects in the Java API. See also Application Metrics Programming Model

While vendors are required to provide a `/metrics` endpoint, as described in this section, it is permissible for implementations to be configurable to run without the endpoint in cases where the metrics capability is not wanted or a different monitoring backend is in use that does not require the endpoint.

Prometheus / OpenMetrics formats

The REST API must respond to GET requests with data formatted according to the Prometheus text-based exposition format, version 0.0.4 (hereafter Prometheus format). For details of how to format metrics data in this format, see Prometheus format.

Implementations may additionally provide the ability to respond to GET requests with data formatted according to the OpenMetrics exposition format, version 1.0 (hereafter OpenMetrics format). For details on how to format metrics data in this format, see OpenMetrics format.

This section provides the details of how to map from the Gauge, Counter, Timer and Histogram types defined in this specification into appropriate fields in the Prometheus format.

Details of how to format metric names, including conventions, special character mapping and placement of the unit (if provided) in the name, are as described by the Prometheus format and OpenMetrics format documentation.

Quantile values, as used in Histogram and Timer output, should represent recent values (typically from the last 5-10 minutes). If no data is available from that timeframe, the value must be set to NaN.

Gauge

Example Gauge with unit `celsius` in Prometheus format.

```
# HELP current_temperature_celsius The current temperature. ①
# TYPE current_temperature_celsius gauge ②
current_temperature_celsius{mp_scope="application",server="front_office"} 36.2 ③
```

① The description of the gauge, from the `getDescription()` method of the `Metadata` associated to the gauge, must be provided in the HELP line

② The type of the metric, in this case `gauge`, must be shown in the TYPE line

③ The value specified must be the value of the gauge's `getValue()` method. Tags, if provided, are included in brackets separated by commas.

Counter

Example Counter with unit `events` in Prometheus format.

```
# HELP messages_processed_events_total Number of messages handled ①
# TYPE messages_processed_events_total counter ②
messages_processed_events_total{mp_scope="application"} 1.0 ③
```

① The description of the counter must be provided in the HELP line

② The type of the metric, in this case `counter`, must be shown in the TYPE line

③ The value specified must be the value of the counter's `getCount()` method. Tags, if provided, are included in brackets separated by commas. By convention, `_total` should be added to the end of the counter name.

Histogram

Example Histogram with unit `meters` in Prometheus format.

```
# HELP distance_to_hole_meters_max Distance of golf ball to hole ①
# TYPE distance_to_hole_meters_max gauge ②
distance_to_hole_meters_max{mp_scope="golf_stats"} 12.722726616315509 ③
# HELP distance_to_hole_meters Distance of golf ball to hole ①
# TYPE distance_to_hole_meters summary ②
distance_to_hole_meters{mp_scope="golf_stats",quantile="0.5"} 2.8748779296875 ③
distance_to_hole_meters{mp_scope="golf_stats",quantile="0.75"} 4.4998779296875 ③
distance_to_hole_meters{mp_scope="golf_stats",quantile="0.95"} 7.9998779296875 ③
distance_to_hole_meters{mp_scope="golf_stats",quantile="0.98"} 9.4998779296875 ③
distance_to_hole_meters{mp_scope="golf_stats",quantile="0.99"} 11.9998779296875 ③
distance_to_hole_meters{mp_scope="golf_stats",quantile="0.999"} 12.9998779296875 ③
distance_to_hole_meters_count{mp_scope="golf_stats"} 487.0 ③
distance_to_hole_meters_sum{mp_scope="golf_stats"} 1569.3785694223322 ③
```

`Histogram` output is comprised of a maximum section and a summary section.

① The description of the histogram must be provided on the HELP lines for the maximum and summary

② The type of the metrics, in this case `gauge` (for the maximum) and `summary` for the summary. The `summary` type is comprised of the count, sum and multiple quantile values.

③ The value of each metric included in the output is described in the table below. Tags, if provided, are included in brackets separated by commas. Percentile metrics include a `quantile` label that is merged with the metric's tags.

Table 2. Prometheus format mapping for a Histogram metric

Suffix{label}	TYPE	Value (Histogram method)	Units
`<units>_max`	Gauge	`getSnapshot().getMax()`	<units>
`<units>{quantile="0.5"}`	Summary	`getSnapshot().getValue(0.5)`	<units>

Suffix{label}	TYPE	Value (Histogram method)	Units
<units>{quantile="0.75"}	Summary	getSnapshot().getValue(0.75)	<units>
<units>{quantile="0.95"}	Summary	getSnapshot().getValue(0.95)	<units>
<units>{quantile="0.98"}	Summary	getSnapshot().getValue(0.98)	<units>
<units>{quantile="0.99"}	Summary	getSnapshot().getValue(0.99)	<units>
<units>{quantile="0.999"}	Summary	getSnapshot().getValue(0.999)	<units>
<units>_count	Summary	getCount()	<units>
<units>_sum	Summary	getSum()	<units>

Timer

Example Timer in Prometheus format. Timers use `seconds` as the unit.

```
# HELP myClass_myMethod_seconds duration of myMethod ①
# TYPE myClass_myMethod_seconds summary ②
myClass_myMethod_seconds{mp_scope="vendor",quantile="0.5"} 0.0524288 ③
myClass_myMethod_seconds{mp_scope="vendor",quantile="0.75"} 0.0524288 ③
myClass_myMethod_seconds{mp_scope="vendor",quantile="0.95"} 0.054525952 ③
myClass_myMethod_seconds{mp_scope="vendor",quantile="0.98"} 0.054525952 ③
myClass_myMethod_seconds{mp_scope="vendor",quantile="0.99"} 0.054525952 ③
myClass_myMethod_seconds{mp_scope="vendor",quantile="0.999"} 0.054525952 ③
myClass_myMethod_seconds_count{mp_scope="vendor"} 100.0 ③
myClass_myMethod_seconds_sum{mp_scope="vendor"} 5.310349419 ③
# HELP myClass_myMethod_seconds_max duration of myMethod ①
# TYPE myClass_myMethod_seconds_max gauge ②
myClass_myMethod_seconds_max{mp_scope="vendor"} 0.05507899 ③
```

`Timer` output is comprised of a maximum section and a summary section.

① The description of the timer must be provided on the HELP lines for the maximum and summary

② The type of the metrics, in this case `gauge` (for the maximum) and `summary` for the summary. The `summary` type is comprised of the count, sum and multiple quantile values.

③ The value of each metric included in the output is described in the table below. Tags, if provided, are included in brackets separated by commas. Percentile metrics include a `quantile` label that is merged with the metric's tags.

Table 3. Prometheus format mapping for a Timer metric

Suffix{label}	TYPE	Value (Timer method)	Units
max_seconds	Gauge	getSnapshot().getMax()	SECONDS[1]
seconds{quantile="0.5"}	Summary	getSnapshot().getValue(0.5)	SECONDS[1]
seconds{quantile="0.75"}	Summary	getSnapshot().getValue(0.75)	SECONDS[1]
seconds{quantile="0.95"}	Summary	getSnapshot().getValue(0.95)	SECONDS[1]

Suffix{label}	TYPE	Value (Timer method)	Units
`seconds{quantile="0.98"}`	Summary	`getSnapshot().getValue(0.98)`	SECONDS[1]
`seconds{quantile="0.99"}`	Summary	`getSnapshot().getValue(0.99)`	SECONDS[1]
`seconds{quantile="0.999"}`	Summary	`getSnapshot().getValue(0.999)`	SECONDS[1]
`seconds_count`	Summary	`getCount()`	SECONDS[1]
`seconds_sum`	Summary	`getElapsedTime()`	SECONDS[1]

[1] The implementation is expected to convert the result returned by the `Timer` into seconds

Security

It must be possible to secure the endpoints via the usual means. The definition of 'usual means' is in this version of the specification implementation specific.

In case of a secured endpoint, accessing `/metrics` without valid credentials must return a `401 Unauthorized` header.

A server SHOULD implement TLS encryption by default.

It is allowed to ignore security for trusted origins (e.g. localhost)

Base Metrics

Base metrics is an optional list of metrics that vendors may implement in whole or in part. These metrics are exposed under `/metrics/base`.

The following is a list of base metrics. All metrics are singletons and have `Multi:` set to `false` unless otherwise stated. Visit Metadata for the meaning of each key

Some virtual machines can not provide the data required for some of the base metrics. Vendors should either use other metrics that are close enough as substitute or not fill these base metrics at all.

General JVM Stats

UsedHeapMemory

Name	memory.usedHeap
Type	Gauge
Unit	Bytes
Description	Displays the amount of used heap memory in bytes.
MBean	java.lang:type=Memory/HeapMemoryUsage#used

CommittedHeapMemory

Name	memory.committedHeap
Type	Gauge
Unit	Bytes
Description	Displays the amount of memory in bytes that is committed for the Java virtual machine to use. This amount of memory is guaranteed for the Java virtual machine to use.
MBean	java.lang:type=Memory/HeapMemoryUsage#committed
Notes	Also from JSR 77

MaxHeapMemory

Name	memory.maxHeap
Type	Gauge
Unit	Bytes

Description	Displays the maximum amount of heap memory in bytes that can be used for memory management. This attribute displays -1 if the maximum heap memory size is undefined. This amount of memory is not guaranteed to be available for memory management if it is greater than the amount of committed memory. The Java virtual machine may fail to allocate memory even if the amount of used memory does not exceed this maximum size.
MBean	java.lang:type=Memory/HeapMemoryUsage#max

GCCount

Name	gc.total
Type	Counter
Unit	None
Multi	true
Tags	{name=%s}
Description	Displays the total number of collections that have occurred. This attribute lists -1 if the collection count is undefined for this collector.
MBean	java.lang:type=GarbageCollector,name=%s/CollectionCount
Notes	There can be multiple garbage collectors active that are assigned to different memory pools. The %s should be substituted with the name of the garbage collector.

GCTime - Approximate accumulated collection elapsed time in ms

Name	gc.time
Type	Gauge
Unit	Seconds
Multi	true
Tags	{name=%s}
Description	Displays the approximate accumulated collection elapsed time in seconds. This attribute displays -1 if the collection elapsed time is undefined for this collector. The Java virtual machine implementation may use a high resolution timer to measure the elapsed time. This attribute may display the same value even if the collection count has been incremented if the collection elapsed time is very short.
MBean	java.lang:type=GarbageCollector,name=%s/CollectionTime
Notes	There can be multiple garbage collectors active that are assigned to different memory pools. The %s should be substituted with the name of the garbage collector. The MicroProfile Metrics runtime will need to convert the metric's value to seconds if the value is provided in a different unit.

JVM Uptime - Up time of the Java Virtual machine

Name	jvm.uptime
Type	Gauge
Unit	Seconds
Description	Displays the time elapsed since the start of the Java virtual machine in seconds.
MBean	java.lang:type=Runtime/Uptime
Notes	Also from JSR 77. The MicroProfile Metrics runtime will need to convert the metric's value to seconds if the value is provided in a different unit.

Thread JVM Stats

ThreadCount

Name	thread.count
Type	Gauge
Unit	None
Description	Displays the current number of live threads including both daemon and non-daemon threads
MBean	java.lang:type=Threading/ThreadCount

DaemonThreadCount

Name	thread.daemon.count
Type	Gauge
Unit	None
Description	Displays the current number of live daemon threads.
MBean	java.lang:type=Threading/DaemonThreadCount

PeakThreadCount

Name	thread.max.count
Type	Gauge
Unit	None
Description	Displays the peak live thread count since the Java virtual machine started or peak was reset. This includes daemon and non-daemon threads.
MBean	java.lang:type=Threading/PeakThreadCount

Thread Pool Stats

ActiveThreads

Name	threadpool.activeThreads
Type	Gauge
Unit	None
Multi	true
Tags	{pool=%s}
Description	Number of active threads that belong to a specific thread pool.
Notes	The %s should be substituted with the name of the thread pool. This is a vendor specific attribute/operation that is not defined in java.lang.

PoolSize

Name	threadpool.size
Type	Gauge
Unit	None
Multi	true
Tags	{pool=%s}
Description	The size of a specific thread pool.
Notes	The %s should be substituted with the name of the thread pool. This is a vendor specific attribute/operation that is not defined in java.lang.

ClassLoading JVM Stats

LoadedClassCount

Name	classloader.loadedClasses.count
Type	Gauge
Unit	None
Description	Displays the number of classes that are currently loaded in the Java virtual machine.
MBean	java.lang:type=ClassLoading/LoadedClassCount

TotalLoadedClassCount

Name	classloader.loadedClasses.total
Type	Counter
Unit	None
Description	Displays the total number of classes that have been loaded since the Java virtual machine has started execution.
MBean	java.lang:type=ClassLoading/TotalLoadedClassCount

UnloadedClassCount

Name	classloader.unloadedClasses.total
Type	Counter
Unit	None
Description	Displays the total number of classes unloaded since the Java virtual machine has started execution.
MBean	java.lang:type=ClassLoading/UnloadedClassCount

Operating System

AvailableProcessors

Name	cpu.availableProcessors
Type	Gauge
Unit	None
Description	Displays the number of processors available to the Java virtual machine. This value may change during a particular invocation of the virtual machine.
MBean	java.lang:type=OperatingSystem/AvailableProcessors

SystemLoadAverage

Name	cpu.systemLoadAverage
Type	Gauge
Unit	None
Description	Displays the system load average for the last minute. The system load average is the sum of the number of runnable entities queued to the available processors and the number of runnable entities running on the available processors averaged over a period of time. The way in which the load average is calculated is operating system specific but is typically a damped time-dependent average. If the load average is not available, a negative value is displayed. This attribute is designed to provide a hint about the system load and may be queried frequently. The load average may be unavailable on some platforms where it is expensive to implement this method.
MBean	java.lang:type=OperatingSystem/SystemLoadAverage

ProcessCpuLoad

Name	cpu.processCpuLoad
Type	Gauge
Unit	Percent

Description	Displays the "recent cpu usage" for the Java Virtual Machine process
MBean	java.lang:type=OperatingSystem (com.sun.management.UnixOperatingSystemMXBean for Oracle Java, similar one exists for IBM Java: com.ibm.lang.management.ExtendedOperatingSystem) Note: This is a vendor specific attribute/operation that is not defined in java.lang

ProcessCpuTime

Name	cpu.processCpuTime
Type	Gauge
Unit	Seconds
Description	Displays the CPU time used by the process on which the Java virtual machine is running in seconds.
MBean	java.lang:type=OperatingSystem (com.sun.management.UnixOperatingSystemMXBean for Oracle Java, similar one exists for IBM Java: com.ibm.lang.management.ExtendedOperatingSystem) Note: This is a vendor specific attribute/operation that is not defined in java.lang. The MicroProfile Metrics runtime will need to convert the metric's value to seconds if the value is provided in a different unit.

REST

The MicroProfile Metrics runtime may track metrics from RESTful resource method calls during runtime (ie. GET, POST, PUT, DELETE, OPTIONS, PATCH, HEAD). It is up to the implementation to decide how to enable the REST metrics.

Mapped and Unmapped Exceptions

The metrics defined below will treat a REST request that ends in a mapped exception or an unmapped exception differently. For the MicroProfile Metrics runtime, mapped exceptions and *succesful* REST requests should be considered and handled the same way. This is because mapped exceptions are expected by the developer and may then be handled appropriately as part of the application's expected behviour. Unmapped exceptions on the other hand are unexpected and can skew metric data if its' respective REST request is recorded. To avoid contaminating the metric values with these *unsuccesful* REST requests, the below metrics may omit tracking a REST request that ends with an unmapped exception. There are also metrics that purposely track REST requests that end with an unmapped exception.

RESTRequest

Name	REST.request
Type	Timer

Unit	None
Multi	true
Tags	{class=%s1,method=%s2}
Description	The number of invocations and total response time of this RESTful resource method since the start of the server. The metric will not record the elapsed time nor count of a REST request if it resulted in an **unmapped** exception. Also tracks the highest recorded time duration and the 50th, 75th, 95th, 98th, 99th and 99.9th percentile.
Notes	With an asynchronous request the **timing** that is tracked by the REST metric must incorporate the time spent by the asynchronous call. The `%s1` should be substituted with the fully qualified name of the RESTful resource class. The `%s2` should be substituted with the name of the RESTful resource method and appended with its parameter types using an underscore `_`. Multiple parameter types are appended one after another (e.g. `<methodName>_<paramType1>_<paramType2>`). Parameter type formatting rules: - The paramter types are fully qualified (e.g. `java.lang.Object`). - Array paramter types will be formatted as `paramType[]` (e.g `java.lang.Object[]`). - A Vararg parameter will be treated as an array. - Generics will be ignored. For example `List<String>` will be formatted as `java.util.List`.

RESTRequestUnmappedExceptions

Name	REST.request.unmappedException.total
Type	Counter
Unit	None
Multi	true
Tags	{class=%s1,method=%s2}
Description	The total number of unmapped exceptions that occur from this RESTful resouce method since the start of the server.

Notes	The `%s1` should be substituted with the fully qualified name of the RESTful resource class. The `%s2` should be substituted with the name of the RESTful resource method and appended with its parameter types using an underscore `_`. Multiple parameter types are appended one after another (e.g. `<methodName>_<paramType1>_<paramType2>`). Parameter type formatting rules: - The paramter types are fully qualified (e.g. `java.lang.Object`). - If the implementation supports array parameters, array parameter types will be formatted as `paramType[]` (e.g `java.lang.Object[]`). - A Vararg parameter will be treated as an array. - Generics will be ignored. For example `List<String>` will be formatted as `java.util.List`.

For example given the following RESTful resource:

```
package org.eclipse.microprofile.metrics.demo;

@ApplicationScoped
public class RestDemo {

  @POST
  public void postMethod(Object o, String... s){
      ...
  }
}
```

The OpenMetrics formatted rest metrics would be:

```
# TYPE REST_request_seconds_max gauge
REST_request_seconds_max{class="org.eclipse.microprofile.metrics.demo.RestDemo",method
="postMethod_java.lang.Object_java.lang.String[]",mp_scope="base"} 1.0
# TYPE REST_request_seconds summary
REST_request_seconds{class="com.ibm.metrics.demo.MyMetrics",method="postMethod_java.la
ng.Object_java.lang.String[]",mp_scope="base",quantile="0.5"} 0.99999744
REST_request_seconds{class="com.ibm.metrics.demo.MyMetrics",method="postMethod_java.la
ng.Object_java.lang.String[]",mp_scope="base",quantile="0.75"} 0.99999744
REST_request_seconds{class="com.ibm.metrics.demo.MyMetrics",method="postMethod_java.la
ng.Object_java.lang.String[]",mp_scope="base",quantile="0.95"} 0.99999744
REST_request_seconds{class="com.ibm.metrics.demo.MyMetrics",method="postMethod_java.la
ng.Object_java.lang.String[]",mp_scope="base",quantile="0.98"} 0.99999744
REST_request_seconds{class="com.ibm.metrics.demo.MyMetrics",method="postMethod_java.la
ng.Object_java.lang.String[]",mp_scope="base",quantile="0.99"} 0.99999744
REST_request_seconds{class="com.ibm.metrics.demo.MyMetrics",method="postMethod_java.la
ng.Object_java.lang.String[]",mp_scope="base",quantile="0.999"} 0.99999744
REST_request_seconds_count{class="com.ibm.metrics.demo.MyMetrics",method="postMethod_j
ava.lang.Object_java.lang.String[]",mp_scope="base"} 1.0
REST_request_seconds_sum{class="com.ibm.metrics.demo.MyMetrics",method="postMethod_jav
a.lang.Object_java.lang.String[]",mp_scope="base"} 1.0
# TYPE REST_request_unmappedException_total counter
REST_request_unmappedException_total{class="org.eclipse.microprofile.metrics.demo.Rest
Demo",method="postMethod_java.lang.Object_java.lang.String[]",mp_scope="base"} 0
```

Application Metrics Programming Model

MicroProfile Metrics provides a way to register Application-specific metrics to allow applications to expose metrics in the *application* scope (see Scopes for the description of scopes).

Metrics and their metadata are added to a *Metric Registry* upon definition and can afterwards have their values set and retrieved via the Java-API and also be exposed via the REST-API (see Exposing metrics via REST API).

Implementors of this specification can use the Java API to also expose metrics for *base* and *vendor* scope by using the respective Metric Registry.

There are two options for registering metrics. The easier one is using annotations - the metrics declared by annotations will be automatically added to the registry when the application starts. In some cases, however, for example when the full list of required metrics is not known in advance, or when it is too large, it might be necessary to interact with the registry programmatically and create new metrics dynamically at runtime. Both approaches can also be combined.

Example set-up of a Gauge metric by an annotation. No unit is given, so `MetricUnits.NONE` *is used, an explicit name is provided*

```
@Gauge(unit = MetricUnits.NONE, name = "queueSize")
public int getQueueSize() {
  return queue.size;
}
```

- NOTE: There are no hard limits on the number of metrics, but it is often not a good practice to create a huge number of metrics, because the downstream time series databases that need to store the metrics may not deal well with this amount of data.

Responsibility of the MicroProfile Metrics implementation

- The implementation must scan the application at deploy time for Annotations and register the Metrics along with their metadata in the *application* MetricsRegistry. This does not apply to gauges, they can be registered lazily when the declaring bean is being instantiated.
- The implementation must watch the annotated objects and update internal data structures when the values of the annotated objects change. The value of a `Gauge` is recomputed each time a client requests the value.
- The implementation must expose the values of the objects registered in the MetricsRegistry via REST-API as described in Exposing metrics via REST API.
- Metrics registered via non-annotations API need their values be set via updates from the application code.
- The implementation must detect if multiple annotations declare the same gauge (with the same `MetricID`) and throw an `IllegalArgumentException` if such duplicate exists

    - See reusing of metrics for more details.
- The implementation must reject metrics upon registration if the metadata information being registered is not equivalent to the metadata information that has already been registered under the given metric name (if it already exists).
    - All metrics of a given metric name must be associated with the same metadata information.
    - The implementation must throw an `IllegalArgumentException` when the metric is rejected.
- The implementation must reject metrics upon registration if the set of tag names specified is not the same as the set of tag names used in prior registrations of metrics with the same metric name.
    - The implementation must throw an `IllegalArgumentException` when the metric is rejected.
- The implementation must throw an `IllegalStateException` if an annotated metric is invoked, but the metric no longer exists in the MetricRegistry. This applies to the following annotations : @Timed, @Counted
- The implementation must make sure that metric registries are thread-safe, in other words, concurrent calls to methods of `MetricRegistry` must not leave the registry in an inconsistent state.

Base Package

All Java-Classes are in the top-level package `org.eclipse.microprofile.metrics` or one of its sub-packages.

Annotations

All Annotations are in the `org.eclipse.microprofile.metrics.annotation` package

These annotations include interceptor bindings as defined by the Java Interceptors specification.

CDI leverages the Java Interceptors specification to provide the ability to associate interceptors to beans via typesafe interceptor bindings, as a means to separate cross-cutting concerns, like Metrics annotations instrumentation, from the application business logic.

Both the Java Interceptors and CDI specifications set restrictions about the type of bean to which an interceptor can be bound.

That implies only *managed beans* whose bean types are *proxyable* can be instrumented using the Metrics annotations.

The following Annotations exist, see below for common fields:

Annotation	Applies to	Description	Default Unit
@Counted	M, C, T	Denotes a counter, which counts the invocations of the annotated object.	MetricUnits.NONE
@Gauge	M	Denotes a gauge, which samples the value of the annotated object.	*no default*, must be supplied by the user
@Metric	F, P	An annotation that contains the metadata information when requesting a metric to be injected.	MetricUnits.NONE
@Timed	M, C, T	Denotes a timer, which tracks duration of the annotated object.	MetricUnits.NANOSECONDS[1]

(C=Constructor, F=Field, M=Method, P=Parameter, T=Type)

[1] Prometheus output will always display timer values in *seconds*. When using the `Timer` API to retrieve the `Snapshot`, the values returned from the `Snapshot` will be in nanoseconds.

Annotation	Description	Default
@RegistryScope	Indicates the scope of Metric Registry to inject when injecting a MetricRegistry.	*application* (scope)
@RegistryType	Qualifies the scope of Metric Registry to inject when injecting a MetricRegistry. **Note: This is deprecated. Please use @RegistryScope**	*application* (scope)

Fields

All annotations (Except `RegistryScope` and `RegistryType`) have the following fields that correspond to the metadata fields described in Metadata.

`String name`

Optional. Sets the name of the metric. If not explicitly given the name of the annotated object is used.

`boolean absolute`

If `true`, uses the given name as the absolute name of the metric. If `false`, prepends the package name and class name before the given name. Default value is `false`.

`String description`

Optional. A description of the metric.

`String unit`

Unit of the metric. For `@Gauge` no default is provided. Check the `MetricUnits` class for a set of pre-defined units.

`String scope`

Optional. The `MetricRegistry` scope that this metric belongs to. Default value is `application`.

Implementors are encouraged to issue warnings in the server log if metadata is missing. Implementors MAY stop the deployment of an application if Metadata is missing.

Annotated Naming Convention

Annotated metrics are registered into the *application* `MetricRegistry` with the name computed using the rules in the following tables.

If the metric annotation is placed on a method or field:

	`name` is specified	`name` is not specified
`absolute=true`	Value of the `name` argument	Name of the annotated element
`absolute=false`	`<canonical-name-of-declaring-class>.<value-of-name-argument>`	`<canonical-name-of-declaring-class>.<name-of-element>`

If the metric annotation is placed on a class, then for each method (including constructors), the metric name will be:

	`name` is specified	`name` is not specified
`absolute=true`	`<value-of-name-argument>.<name-of-the-method>`	`<short-name-of-class>.<name-of-the-method>`
`absolute=false`	`<package-of-the-declaring-class>.<value-of-name-argument>.<name-of-the-method>`	`<canonical-name-of-the-declaring-class>.<name-of-the-method>`

In case of constructors, "name of the method" is the short name of the declaring class.

Examples of metric names when metric annotations are applied to beans

```
package com.example;

import jakarta.inject.Inject;
import org.eclipse.microprofile.metrics.Counter;
import org.eclipse.microprofile.metrics.annotation.Metric;

public class Colours {

    @Counted
    public void red() {
        // ...
    }

    @Counted(name="blueCount")
    public void blue() {
        // ...
    }

    @Counted(name="greenCount", absolute=true)
    public void green() {
        // ...
    }

    @Counted(absolute=true)
    public void yellow() {
        // ...
    }

}
```

The above bean would produce the following entries in the `MetricRegistry`

```
com.example.Colours.red
com.example.Colours.blueCount
greenCount
yellow
```

Examples of metric names when `@Inject` is used together with `@Metric`

```
package com.example;

import jakarta.inject.Inject;
import org.eclipse.microprofile.metrics.Counter;
import org.eclipse.microprofile.metrics.annotation.Metric;

public class Colours {

  @Inject
  @Metric
  Counter redCount;

  @Inject
  @Metric(name="blue")
  Counter blueCount;

  @Inject
  @Metric(absolute=true)
  Counter greenCount;

  @Inject
  @Metric(name="purple", absolute=true)
  Counter purpleCount;
}
```

The above bean would produce the following entries in the `MetricRegistry`

```
com.example.Colours.redCount
com.example.Colours.blue
greenCount
purple
```

@Counted

An annotation for marking a method, constructor, or type as a counter.

The implementation must support the following annotation targets:

- `CONSTRUCTOR`
- `METHOD`
- `TYPE`

This annotation has changed in MicroProfile Metrics 2.0: Counters now always increase monotonically upon invocation.

If the metric no longer exists in the `MetricRegistry` when the annotated element is invoked then an

`IllegalStateException` will be thrown.

The following lists the behavior for each annotation target.

CONSTRUCTOR

When a constructor is annotated, the implementation must register a counter for the constructor using the Annotated Naming Convention. The counter is increased by one when the constructor is invoked.

Example of an annotated constructor

```
@Counted
public CounterBean() {
}
```

METHOD

When a non-private method is annotated, the implementation must register a counter for the method using the Annotated Naming Convention. The counter is increased by one when the method is invoked.

Example of an annotated method

```
@Counted
public void run() {
}
```

TYPE

When a type/class is annotated, the implementation must register a counter for each of the constructors and non-private methods using the Annotated Naming Convention. The counters are increased by one when the corresponding constructor/method is invoked.

Example of an annotated type/class

```
@Counted
public class CounterBean {

  public void countMethod1() {}
  public void countMethod2() {}

}
```

@Gauge

An annotation for marking a method as a gauge. No default `MetricUnit` is supplied, so the `unit` must always be specified explicitly.

The implementation must support the following annotation target:

- `METHOD`

The following lists the behavior for each annotation target.

METHOD

When a non-private method is annotated, the implementation must register a gauge for the method using the Annotated Naming Convention. The gauge value and type is equal to the annotated method return value and type.

Example of an annotated method

```
@Gauge(unit = MetricUnits.NONE)
public long getValue() {
  return value;
}
```

@Timed

An annotation for marking a constructor or method of an annotated object as timed. The metric of type Timer tracks how frequently the annotated object is invoked, and tracks how long it took the invocations to complete. The data is aggregated to calculate duration statistics and throughput statistics.

The implementation must support the following annotation targets:

- `CONSTRUCTOR`
- `METHOD`
- `TYPE`

If the metric no longer exists in the `MetricRegistry` when the annotated element is invoked then an `IllegalStateException` will be thrown.

The following lists the behavior for each annotation target.

CONSTRUCTOR

When a constructor is annotated, the implementation must register a timer for the constructor using the Annotated Naming Convention. Each time the constructor is invoked, the execution will be timed.

Example of an annotated constructor

```
@Timed
public TimedBean() {
}
```

METHOD

When a non-private method is annotated, the implementation must register a timer for the method using the Annotated Naming Convention. Each time the method is invoked, the execution will be timed.

Example of an annotated method

```
@Timed
public void run() {
}
```

TYPE

When a type/class is annotated, the implementation must register a timer for each of the constructors and non-private methods using the Annotated Naming Convention. Each time a constructor/method is invoked, the execution will be timed with the corresponding timer.

Example of an annotated type/class

```
@Timed
public class TimedBean {

  public void timedMethod1() {}
  public void timedMethod2() {}

}
```

@Metric

An annotation requesting that a metric should be injected or registered.

The implementation must support the following annotation targets:

- `FIELD`
- `PARAMETER`

The following lists the behavior for each annotation target.

FIELD

When a metric injected field is annotated, the implementation must provide the registered metric with the given name (using the Annotated Naming Convention) if the metric already exists. If no metric exists with the given name then the implementation must produce and register the requested metric.

Gauges are an exception to this rule, because it could happen that an annotated gauge is not registered yet when the reference to it is being injected. In that case, the implementation must inject a proxy gauge implementation which forwards `getValue()` calls to the actual gauge, if the

actual gauge already exists. If `getValue()` is called on the proxy gauge and the actual gauge still does not exist in the registry, `getValue()` will return null.

Example of an injected field

```
@Inject
@Metric(name = "applicationCount")
Counter count;
```

PARAMETER

When a metric parameter is annotated, the implementation must provide the registered metric with the given name (using the Annotated Naming Convention) if the metric already exist. If no metric exists with the given name then the implementation must produce and register the requested metric.

Example of an annotated parameter

```
@Inject
public void init(@Metric(name="instances") Counter instances) {
    instances.inc();
}
```

Usage of CDI stereotypes

If a metric annotation is applied to a bean through a CDI stereotype, the implementation must handle it the same way as if the metric annotation was applied on the target bean directly. Metric names are computed relative to the name and package of the bean itself, not of the stereotype.

Registering metrics dynamically

In addition to declaring metrics via annotations, it is possible to dynamically (un)register metrics by calling methods of a `MetricRegistry` object. Registering metrics dynamically can be useful in some cases, for example, when the final list of metrics is not known in advance (when the application is being coded), or when there are too many similar metrics and it would be more practical to register them in a `for` loop than to introduce lots of annotations in the code. The two approaches can also be combined if necessary.

List of methods of the MetricRegistry related to registering new metrics

Method	Description
`counter(String name)`	Counter with given name and no tags
`counter(String name, Tag… tags)`	Counter with given name and tags
`counter(Metadata metadata)`	Counter from given `Metadata` object

Method	Description
`counter(Metadata metadata, Tag… tags)`	Counter from given `Metadata` object with given tags
`histogram(String name)`	Histogram with given name and no tags
`histogram(String name, Tag… tags)`	Histogram with given name and tags
`histogram(Metadata metadata)`	Histogram from given `Metadata` object
`histogram(Metadata metadata, Tag… tags)`	Histogram from given `Metadata` object with given tags
`timer(String name)`	Timer with given name and no tags
`timer(String name, Tag… tags)`	Timer with given name and tags
`timer(Metadata metadata)`	Timer from given `Metadata` object
`timer(Metadata metadata, Tag… tags)`	Timer from given `Metadata` object with given tags

All metrics in the table above, except the variants of `register`, exhibit the *get-or-create* semantics, so if a compatible metric with the same `MetricID` already exists, the existing one is returned. "Compatible" in this context means that the type and all specified metadata must be equal - else an exception is thrown. If a metric exists under the same name but with different tags, the newly created metric must have all of its metadata equal to the existing metric's metadata.

The `register` method variants exhibit the *create* semantics, that means, if a metric with the same `MetricID` already exists, an exception is thrown. If a metric exists under the same name but with different tags, the newly created metric must have all of its metadata equal to the existing metric's metadata.

Unregistering metrics

While the general recommendation is that metrics live for the whole lifecycle of the application, it is still possible to dynamically remove metrics from metric registries at runtime.

List of methods of the MetricRegistry related to removing metrics

Method	Description
`remove(String name)`	Removes all metrics with the given name
`remove(MetricID metricID)`	Removes the metric with the given `MetricID`, if it exists
`remove(MetricFilter filter)`	Removes all metrics that are accepted by the given `MetricFilter` instance

Metric Registries

The `MetricRegistry` is used to maintain a collection of metrics along with their metadata. There is one shared singleton of the `MetricRegistry` per pre-defined scope (*application*, *base*, and *vendor*).

There is also one shared singleton of the `MetricRegistry` per custom scope. When metrics are registered using annotations and no scope is provided, the metrics are registered in the *application* `MetricRegistry` (and thus the *application* scope).

When injected, the `@RegistryScope` is used to selectively inject one of the `application`, `base`, `vendor` or custom registries. If no *scope* parameter is used, the default `MetricRegistry` returned is the `application` registry.

If using the **deprecated** `@RegistryType`, it will be used as a qualifier to selectively inject one of the `application`, `base` or `vendor` registries. If no qualifier is used, the default `MetricRegistry` returned is the `application` registry. Note that `@RegistryType` is now deprecated. Please use `@RegistryScope` instead.

The `@RegistryScope` annotation and `@RegistryType` qualifer annotation should not be used together for the same `MetricRegistry` injection. The `application`, `base` or `vendor` registry produced by either injection strategies should be the same respective to the scope. That is to say, an injection of a `MetricRegistry` with `@RegistryScope(scope = MetricRegistry.APPLICATION_SCOPE)` will produce the same `MetricRegistry` from using `@RegistryType(type = MetricRegistry.Type.APPLICATION)` and vice-versa.

Implementations may choose to use a Factory class to produce the injectable `MetricRegistry` bean via CDI. See Example Metric Registry Factory. Note: The factory would be an internal class and not exposed to the application.

@RegistryScope

The `@RegistryScope` can be used to retrieve the `MetricRegistry` for a specific scope. The implementation must produce the corresponding `MetricRegistry` specified by the `RegistryScope`.

@RegistryType

The `@RegistryType` can be used to retrieve the `MetricRegistry` for a specific scope. The implementation must produce the corresponding `MetricRegistry` specified by the `RegistryType`.

The implementor can optionally provide a *read_only* copy of the `MetricRegistry` for *base* and *vendor* scopes.

The `@RegistryType` is **deprecated**. Please use `@RegistryScope` instead.

Application Metric Registry

The implementation must produce the *application* `MetricRegistry` when no `RegistryScope` (or `RegistryType`) is provided or when the `RegistryScope` is `application` (i.e. `MetricRegistry.APPLICATION_SCOPE`) or if the **deprecated** `RegistryType` is `application` (i.e. `MetricRegistry.Type.APPLICATION`). Application-defined metrics can also be registered to user-defined scopes

Example of the application injecting the application registry

```
@Inject
MetricRegistry metricRegistry;
```

is equivalent to the following with `@RegistryScope`

```
@Inject
@RegistryScope(scope=MetricRegistry.APPLICATION_SCOPE)
MetricRegistry metricRegistry;
```

or is equivalent to the following with the **deprecated** `@RegistryType`

```
/*
 * @RegistryType is deprecated. Please use @RegistryScope
 */
@Inject
@RegistryType(type=MetricRegistry.Type.APPLICATION)
MetricRegistry metricRegistry;
```

Base Metric Registry

The implementation must produce the *base* `MetricRegistry` when the `RegistryScope` is `base` (i.e. `MetricRegistry.BASE_SCOPE`). The *base* `MetricRegistry` contains any metrics the vendor has chosen to provide from Base Metrics.

Example of the application injecting the base registry using `@RegistryScope`

```
@Inject
@RegistryScope(scope=MetricRegistry.BASE_SCOPE)
MetricRegistry baseRegistry;
```

Example of the application injecting the base registry using the **deprecated** `@RegistryType`

```
@Inject
@RegistryType(type=MetricRegistry.Type.BASE)
MetricRegistry baseRegistry;
```

Vendor Metric Registry

The implementation must produce the *vendor* `MetricRegistry` when the `RegistryScope` is `vendor` (i.e. `MetricRegistry.VENDOR_SCOPE`). The *vendor* `MetricRegistry` must contain any vendor specific metrics.

Example of the application injecting the vendor registry using `@RegistryScope`

```
@Inject
@RegistryScope(scope=MetricRegistry.VENDOR_SCOPE)
MetricRegistry vendorRegistry;
```

Example of the application injecting the vendor registry using the **deprecated** `@RegistryType`

```
@Inject
@RegistryType(type=MetricRegistry.Type.VENDOR)
MetricRegistry vendorRegistry;
```

The implementation must produce the `MetricRegistry` corresponding to the custom-named registry when the `RegistryType` is a custom value. If the custom-named `MetricRegistry` does not yet exist the implementation must create a `MetricRegistry` with the specified name.

Example of the application injecting a custom-named registry

```
@Inject
@RegistryScope(scope="motorguide")
MetricRegistry motorGuideRegistry;
```

Metadata

Metadata is used in MicroProfile-Metrics to provide immutable information about a Metric at registration time. Metadata in the architecture section describes this further.

Therefore `Metadata` is an interface to construct an immutable metadata object. The object can be built via a `MetadataBuilder` with a fluent API.

Example of constucting a `Metadata` object for a Meter and registering it in Application scope

```
Metadata m = Metadata.builder()
    .withName("myMeter")
    .withDescription("Example meter")
    .build();

Meter me = new MyMeterImpl();
metricRegistry.register(m, me, new Tag("colour","blue"));
```

A default implementation `DefaultMetadata` is provided in the API for convenience.

Micrometer Implementations

Vendor implementations are required to implement the REST interfaces detailed in the REST endpoints section of this document, including the `/metrics` endpoint that provides metrics data in Prometheus format, in order to provide metrics to monitoring agents.

In order to achieve this, vendors MAY choose to implement metrics in their products using Micrometer, OpenTelemetry Metrics or another library, but they are not required to do so.

Micrometer Backends

Where a vendor chooses to use Micrometer, they MAY additionally wish to support Micrometer's other monitoring backends, which at the time of writing include:

- AppOptics
- Azure Monitor
- Netflix Atlas
- CloudWatch
- Datadog
- Dynatrace
- Elastic
- Ganglia
- Graphite
- Humio
- Influx/Telegraf
- JMX
- KairosDB
- New Relic
- Prometheus
- SignalFx
- Google Stackdriver
- StatsD
- Wavefront

The `/metrics` REST endpoint provides Prometheus metrics as a pull-based mechanism. A monitoring agent will need to make a request to the endpoint to obtain the metrics data at that point in time. Conversely, the Micrometer backends listed above are typically push-based, so vendor products using the Micrometer backends will be periodically pushing metrics data from the server to the metrics backend.

Recommended setup and configuration for alternative Micrometer backends

The following suggestions are OPTIONAL, and provided with a view of attempting to make configuring Micrometer-based metrics implementations consistent for consumers.

Discoverability

Each Micrometer backend is packaged separately by the Micrometer project in its own .jar file. In order to allow an implementation to push to Graphite, for example, the vendor will either need to provide the io.micrometer:micrometer-registry-graphite jar and its runtime dependencies as part of their product, or enable consumers to add it to their classpath. Where consumers are adding libraries to the classpath, vendors can check for the presence of the appropriate MeterRegistry class. Taking Graphite as an example, the vendor would need to check for the presence of `io.micrometer.graphite.GraphiteMeterRegistry`.

Configuration

Each Micrometer backend has its own Config interface, which requires a `String get(final String propertyName)` method to be implemented. In order to configure the Micrometer backends in a way that is both consistent across all the Micrometer backends and also consistent with MicroProfile itself, it is suggested that the `String get(final String propertyName)` is implemented to obtain the relevant config using MicroProfile config. Micrometer property names are already prefixed with the name of the relevant backend, so it is suggested that a prefix of `mp.metrics.` is added to the property when it is obtained from MicroProfile config.

Enabling a backend

Micrometer backends have many values in their config set by default. It is therefore recommended that backends are not enabled by default, and enabled by setting `mp.metrics.<backend name>.enabled` to `true`, for example:

```
mp.metrics.graphite.enabled = true
```

Example backend setup and configuration

If a vendor is implementing MicroProfile Metrics as a CDI extension, the above can be achieved by registering a Producer for a backend, if the relevant Micrometer registry class is available on the classpath.

The following is provided as an example of a CDI producer for the Graphite backend.

```
public static class GraphiteBackendProducer {

        @Inject
        private Config config;

        @Produces
        @Backend
        public MeterRegistry produce() {
            if (!Boolean.parseBoolean(
                    config.getOptionalValue("mp.metrics.graphite.enabled",
String.class).orElse("false"))) {
                return null;
            }

            return new GraphiteMeterRegistry(new GraphiteConfig() {
                @Override
                public String get(final String propertyName) {
                    return config.getOptionalValue("mp.metrics." + propertyName,
String.class)
                            .orElse(null);
                }
            }, io.micrometer.core.instrument.Clock.SYSTEM);
        }
    }
```

Appendix

Alternatives considered

We addressed some significant questions while creating MicroProfile Metrics v5.0.

API or no API

In light of the increasing prevalence of developer use of APIs from Micrometer and OpenTelemetry, we considered whether we should continue to have a distinct API for MicroProfile Metrics. We decided to continue our path of providing an API for the following reasons:

1. provides an easy-to-use metrics API for application developers
2. provides continuity for the existing MicroProfile Metrics user community
3. provides a MicroProfile-style API (for example, CDI-based annotations), and configurability (MicroProfile Config), for ease of adoption by MicroProfile users
4. ensures compatibility across APIs within the same MicroProfile release

We also considered feedback from an informal poll in which a majority of respondents said they would use a MicroProfile Metrics API, given the other options.

Fixed implementation or vendor-chosen implementation

We considered whether MicroProfile Metrics should require vendors to use a particular metrics library in their implementations. The benefit of requiring a particular metrics library would be the potential for improved consistency across vendors. The benefits of not requiring a particular metrics library would be avoiding MicroProfile potentially overreaching by telling vendors which libraries to use, and leaving flexibility for vendors to change their implementation in the future if needed. Ultimately, we decided to not require a specific metrics library to be used in the implementation. Vendors may choose to implement using Micrometer libraries, OpenTelemetry libraries, Dropwizard libraries, custom code, or anything else they choose.

References

Micrometer

OpenTelemetry Metrics

Dropwizard Metrics 3.2.3

CDI extension for Dropwizard Metrics 1.4.0

HTTP return codes

UoM, JSR 363

Metrics 2.0

Example configuration format for base and vendor-specific data

The following is an example configuration in YAML format.

```
base:
  - name: "thread-count"
    mbean: "java.lang:type=Threading/ThreadCount"
    description: "Number of currently deployed threads"
    unit: "none"
    type: "gauge"
  - name: "peak-thread-count"
    mbean: "java.lang:type=Threading/PeakThreadCount"
    description: "Max number of threads"
    unit: "none"
    type: "gauge"
  - name: "total-started-thread-count"
    mbean: "java.lang:type=Threading/TotalStartedThreadCount"
    description: "Number of threads started for this server"
    unit: "none"
    type: "counter"
  - name: "max-heap"
    mbean: "java.lang:type=Memory/HeapMemoryUsage#max"
    description: "Number of threads started for this server"
    unit: "bytes"
    type: "counter"
    tags: "kind=memory"

vendor:
  - name: "msc-loaded-modules"
    mbean: "jboss.modules:type=ModuleLoader,name=BootModuleLoader-2/LoadedModuleCount"
    description: "Number of loaded modules"
    unit: "none"
    type: "gauge"
```

This configuration can be backed into the runtime or be provided via an external configuration file.

Example Metric Registry Factory

Sample skeleton factory class to produce MetricRegistry via CDI

```
@ApplicationScoped
public class MetricRegistryFactory {

    @Produces
    @Default
    public MetricRegistry getMetricRegistry(InjectionPoint ip) {

        RegistryScope registryTypeAnnotation = ip.getAnnotated().getAnnotation
(RegistryScope.class);

        if (registryTypeAnnotation == null) {
            return getOrCreate(MetricRegistry.APPLICATION_SCOPE);
        } else {
            String annoScope = registryTypeAnnotation.scope();
            return getOrCreate(annoScope);
        }
    }

    @Produces
    @RegistryType(type = MetricRegistry.Type.APPLICATION)
    public MetricRegistry getApplicationRegistry() {
        return getOrCreate(MetricRegistry.Type.APPLICATION);
    }

    @Produces
    @RegistryType(type = MetricRegistry.Type.BASE)
    public MetricRegistry getBaseRegistry() {
        return getOrCreate(MetricRegistry.Type.BASE);
    }

    @Produces
    @RegistryType(type = MetricRegistry.Type.VENDOR)
    public MetricRegistry getVendorRegistry() {
        return getOrCreate(MetricRegistry.Type.VENDOR);
    }

}
```

Migration hints

To version 5.0

SimpleTimer / @SimplyTimed

The `SimpleTimer` class and `@SimplyTimed` annotation have been removed. This change was made to make it possible to implement the spec using commonly used metrics libraries that lack a similar metric type.

Use `Timer` class or `@Timed` annotation instead. Alternatively, you can create your own `Gauge` to track the total time and your own `Counter` to track the total number of hits of something you want to time.

ConcurrentGauge / @ConcurrentGauge

The `ConcurrentGauge` class and `@ConcurrentGauge` annotation have been removed. This change was made to make it possible to implement the spec using commonly used metrics libraries that lack a similar metric type.

Use `Gauge` class or `@Gauge` annotation instead. A `Gauge` allows you to track a value that may go up or down over time. If you need to track the recent maximum or minimum with precision (as was handled by a `ConcurrentGauge`), create a separate `Gauge` for each of those statistics, in addition to the `Gauge` to track the current value of what you are observing.

Meter / @Metered

The `Meter` class and `@Metered` annotation have been removed. This change was made to make it possible to implement the spec using commonly used metrics libraries that lack a similar metric type.

Use `Counter` class or `@Counted` annotation instead. Tools, such as Prometheus, are able to compute the rate of increase of an observed metric over a specified period of time.

Snapshot

The `Snapshot` class has been modified to avoid restricting the list of percentiles to a fixed set of percentile values. This change was made in anticipation of making the list of percentiles be configurable in the future. As in prior releases, the `Timer` and `Histogram` classes still track the 50th, 75th, 95th, 98th, 99th, and 99.9th percentiles in the corresponding `Snapshot`.

Use `snapshot.percentileValues()` method, then iterate over the returned array of `PercentileValue` objects to find the value at the specific percentile you're interested in.

Metric names

The `base_`, `vendor_` and `application_` prefixes for metric names that were used in prior releases have been replaced by a tag named `mp_scope` with value `base`, `vendor`, or `application` (you can also register metrics with custom scopes).

When using the Prometheus format output from the `/metrics` endpoint, use `metric_name{mp_scope="scopeValue",⋯}` instead of `scopeValue_metric_name{⋯}` where `metric_name` is the Prometheus-formatted name of your metric and `scopeValue` is one of `base`, `vendor`, `application` or a custom value.

Release Notes

Changes in 5.0

Incompatible Changes

- This release aligns with Jakarta EE 10, so it won't work with earlier versions of Jakarta or Java EE

Breaking changes

- Removed SimpleTimer class and SimplyTimed annotation
- Removed ConcurrentGauge class and ConcurrentGauge annotation
- Removed Meter class and Metered annotation
- Removed Metered interface
- Removed MetricType enum
- Updated Timer class
  - Removed `getFifteenMinuteRate()` method
  - Removed `getFiveMinuteRate()` method
  - Removed `getMeanRate()` method
  - Removed `getOneMinuteRate()` method
  - Removed `getStdDev()` method
- Updated MetricRegistry class
  - Removed `register(String name, T metric)` method
  - Removed `register(Metadata metadata, T metric)` method
  - Removed `register(Metadata metadata, T metric, Tag… tags)` method
  - Removed `concurrentGauge(String name)` method
  - Removed `concurrentGauge(String name, Tag… tags)` method
  - Removed `concurrentGauge(MetricID metricID)` method
  - Removed `concurrentGauge(Metadata metadata)` method
  - Removed `concurrentGauge(Metadata metadata, Tag… tags)` method
  - Removed `meter(String name)` method
  - Removed `meter(String name, Tag… tags)` method
  - Removed `meter(MetricID metricID)` method
  - Removed `meter(Metadata metadata)` method
  - Removed `meter(Metadata metadata, Tag… tags)` method
  - Removed `simpleTimer(String name)` method
  - Removed `simpleTimer(String name, Tag… tags)` method

    - Removed `simpleTimer(MetricID metricID)` method
    - Removed `simpleTimer(Metadata metadata)` method
    - Removed `simpleTimer(Metadata metadata, Tag… tags)` method
    - Removed `getConcurrentGauge(MetricID metricID)` method
    - Removed `getConcurrentGauges()` method
    - Removed `getConcurrentGauges(MetricFilter filter)` method
    - Removed `getMeter(MetricID metricID)` method
    - Removed `getMeters()` method
    - Removed `getMeters(MetricFilter filter)` method
    - Removed `getSimpleTimer(MetricID metricID)` method
    - Removed `getSimpleTimers()` method
    - Removed `getSimpleTimers(MetricFilter filter)` method
- Updated DefaultMetadata class
    - Removed displayName from constructor
    - Removed `getDisplayname()` method
    - Removed `displayName()` method
    - Removed metricType from constructor
    - Removed `getType()` method
    - Removed `getTypeRaw()` method
- Updated Metadata class
    - Removed `getDisplayname()` method
    - Removed `displayName()` method
    - Removed `getType()` method
    - Removed `getTypeRaw()` method
- Updated MetadataBuilder class
    - Removed `withDisplayName(String displayName)` method
    - Removed `withType(MetricType type)` method
- Updated Snapshot class
    - Removed `getValue(double quantile)` method
    - Removed `getValues()` method
    - Removed `get75thPercentile()` method
    - Removed `get95thPercentile()` method
    - Removed `get98thPercentile()` method
    - Removed `get999thPercentile()` method
    - Removed `get99thPercentile()` method

    - Removed `getMedian()` method
    - Removed `getMin()` method
    - Removed `getStdDev()` method
    - Modified `size()` method to return long
    - Modified `getMax()` method to return double
- Updated Gauge class
    - can now only work with types that extend Number
- Updated MetricType class
    - Removed `CONCURRENT_GAUGE` enum
    - Removed `METERED` enum
    - Removed `SIMPLE_TIMER` enum

API/SPI Changes

- Updated Snapshot class
    - Added `percentileValues()` method
    - Added Snapshot.PercentileValue inner class
- Deprecated @RegistryType and MetricRegistry.Type (746)

Functional Changes

- Added concept of custom scopes for metrics (677)
    - added tagging of all metrics with mp_scope=value
    - changed /metrics/base to /metrics?scope=base (692)
    - changed /metrics/vendor to /metrics?scope=vendor (692)
    - changed /metrics/application to /metrics?scope=application (692)
    - added /metrics?scope=myScope for custom scoped metrics (677)
    - added ability for applications to add metrics to a custom scope (677)
    - added ability to use custom scope names with @RegistryScope annotation (677)
    - replaced @RegistryType with @RegistryScope (677)
- Other changes
    - removed requirement to convert metrics to base units for Prometheus output
    - changed from prepending scope to the metric name to putting the scope in mp_scope tag
    - clarified that implementations of /metrics endpoint must support Prometheus text-based exposition format, and may also support OpenMetrics exposition format. (678)
    - removed JSON format for /metrics output (685)
    - added restriction to block apps from adding metric IDs with the reserved mp_scope and

mp_app tag names (700)

- changed _app tag name to mp_app (705)
- added mp_scope tag to indicate metric scope
- added configuration recommendations for vendors implementing the API with Micrometer libraries
- added rule that metrics of the same name must all contain the same label set (721)
- changed REST.request metric from SimpleTimer to Timer type
- changed the base metrics to be optional (680)

Changes in 4.0

Incompatible Changes

- This release aligns with Jakarta EE 9.1, so it won't work with earlier versions of Jakarta or Java EE (#639)

Changes in 3.0

Breaking changes

- Removed everything related to reusability from the API code. All metrics are now considered reusable.
- CDI producers annotated with `@Metric` no longer trigger metric registration. If these metrics should be registered, it must be done differently (for example using the `MetricRegistry` methods)
- `MetricRegistry` changed from `abstract class` to `interface`
- Changed `Timer.update(long duration, java.util.concurrent.TimeUnit)` to `Timer.update(java.time.Duration duration)`
- Removed `MetadataBuilder.withOptional*` methods, the remaining `with*` methods do accept `null` value (considered not present) except `withName` which does not accept `null` or `""`
- Changed `Metadata.getDescription()` and `Metadata.getUnit()` to return `String` instead of `Optional<String>` and added `Metadata.description()` and `Metadata.unit()` that return `Optional<String>`

API/SPI Changes

- Updated dependencies scopes and versions to align with Jakarta EE 8
- `MetricRegistry` changed from `abstract class` to `interface`
- Added the `MetricRegistry.getType()` method
- Added the `MetricRegistry.counter(MetricID)` method
- Added the `MetricRegistry.concurrentGauge(MetricID)` method
- Added the `MetricRegistry.gauge(String, Object, Function, Tag[])` method
- Added the `MetricRegistry.gauge(MetricID, Object, Function)` method
- Added the `MetricRegistry.gauge(Metadata, Object, Function, Tag[])` method
- Added the `MetricRegistry.gauge(String, Supplier, Tag[])` method
- Added the `MetricRegistry.gauge(MetricID, Supplier)` method
- Added the `MetricRegistry.gauge(Metadata), Supplier, Tag[])` method
- Added the `MetricRegistry.histogram(MetricID)` method
- Added the `MetricRegistry.meter(MetricID)` method
- Added the `MetricRegistry.timer(MetricID)` method
- Added the `MetricRegistry.simpleTimer(MetricID)` method
- Added the `MetricRegistry.getMetric(MetricID)` method
- Added the `MetricRegistry.getMetric(MetricID metricID, Class)` method
- Added the `MetricRegistry.getCounter(MetricID)` method

- Added the `MetricRegistry.getConcurrentGauge(MetricID)` method
- Added the `MetricRegistry.getGauge(MetricID)` method
- Added the `MetricRegistry.getHistogram(MetricID)` method
- Added the `MetricRegistry.getMeter(MetricID)` method
- Added the `MetricRegistry.getTimer(MetricID)` method
- Added the `MetricRegistry.getSimpleTimer(MetricID)` method
- Added the `MetricRegistry.getMetadata(String)` method
- Added the `MetricRegistry.getMetrics(MetricFilter)` method
- Added the `MetricRegistry.getMetrics(Class, MetricFilter)` method
- Added `SimpleTimer.getMinTimeDuration()` and `SimpleTimer.getMaxTimeDuration()` methods which return a `java.time.Duration` object (#523)
- Timer class updated (#524)
  - Changed `Timer.update(long duration, java.util.concurrent.TimeUnit)` to `Timer.update(java.time.Duration duration)`
  - Added `Timer.getElapsedTime()` which returns `java.time.Duration`
- Removed `MetadataBuilder.withOptional*` methods
- Global tags and `_app` tag are no longer handled automatically by the `MetricID` class, the implementation is expected to add them by itself, for example during metric export
- Added the `Histogram.getSum()` which returns `long` (#597)

Functional Changes

- Simple Timer metrics now track the highest and lowest recorded timing duration of the previous completed minute (#523)
- Timer now exposes total elapsed time duration as a metric value. (#524)
- Clarified that the existing REST metric `REST.request` will not monitor and track a REST request to a REST endpoint if an unmapped exception occurs.
- Introduced a new base REST metric `REST.request.unmappedException.total` that counts the occurrences of unmapped exceptions for each REST endpoint (#533)
- Histogram now exposes the total sum of recorded values as a `sum` value (#597)
  - In JSON format it is exposed as a `sum` value
  - In OpenMetrics format it is exposed as a `sum` value under the `summary` type
- Timer now exposes the `elapsedTime` metric value as `sum` under the `summary` type in OpenMetrics format (#597)

Specification Changes

- Removed the concept of reusability

- CDI producers annotated with `@Metric` no longer trigger metric registration
- Clarified how the implementation must handle metrics applied via CDI stereotypes
- The implementation is required to sanitize `Metadata` passed by the application in cases when it does not contain an explicit type, but the type is implied by the name of the registration method that is being called.
- Clarified that the existing REST metric `REST.request` will not monitor and track a REST request to a REST endpoint if an unmapped exception occurs
- Introduced a new base REST metric `REST.request.unmappedException.total` that counts the occurrences of unmapped exceptions for each REST endpoint (#533)
- Histogram now exposes the total sum of recorded values as a `sum` value (#597)
  - In JSON format it is exposed as a `sum` value
  - In OpenMetrics format it is exposed as a `sum` value under the `summary` type
- Timer now exposes the `elapsedTime` metric value as `sum` under the `summary` type in OpenMetrics format (#597)

TCK enhancement

- Improved TCK - Use newly introduced `MetricRegistry` methods to retrieve single metrics and avoid use of the `getMetrics()` and `getMetadata()` methods

Changes in 2.3

A full list of changes may be found on the MicroProfile Metrics 2.3 Milestone

API/SPI Changes

- Introduced the simple timer (`@SimplyTimed`) metric. (#496)
- Added `withOptional*` methods to the `MetadataBuilder`, they don't fail when null values are passed to them (#464)
- Added the `MetricID.getTagsAsArray()` method to the API. (#457)
- Added the method `MetricType.fromClassName` (#455)

Functional Changes

- Introduced a new base metric derived from RESTful stats into the base scope.
  - `REST.request` : Tracks the total count of requests and total elapsed time spent at the REST endpoint
- Introduced the simple timer (`@SimplyTimed`) metric. (#496)
- The API code no longer requires a correctly configured MP Config implementation to be available at runtime, so it is possible to slim down deployments if MP Config is not necessary (#466)

Specification Changes

- Introduced a new base metric derived from RESTful stats into the base scope.
  - `REST.request` : Tracks the total count of requests and total elapsed time spent at the REST endpoint
- Introduced the simple timer (`@SimplyTimed`) metric. (#496)
- Added ProcessCpuTime as a new optional base metric. (#442)

TCK enhancement

- Improved TCK - Use WebArchive for deployment

Changes in 2.2

A full list of changes may be found on the MicroProfile Metrics 2.2.1 Milestone

API/SPI Changes

- Reverted a problematic change from 2.1 where Gauges were required to return subclasses of `java.lang.Number`

Functional Changes

- Reverted a problematic change from 2.1 where Gauges were required to return subclasses of `java.lang.Number`
- (2.2.1) Added ProcessCpuTime as a new optional base metric. (#480)

Specification Changes

- (2.2.1) Added ProcessCpuTime as a new optional base metric. (#480)

Changes in 2.1

A full list of changes may be found on the MicroProfile Metrics 2.1 Milestone and MicroProfile Metrics 2.1.1 Milestone

API/SPI Changes

- Clarified in the API code that Gauges must return values that extend `java.lang.Number`. [NOTE: this caused issues with backward compatibility and was reverted in 2.2] (#304)
- Added the `reusable(boolean)` method for MetadataBuilder (#407)

Functional Changes

- (2.1.1) Added ProcessCpuTime as a new optional base metric. (#454)
- Clarified in the API code that Gauges must return values that extend `java.lang.Number`. [NOTE: this caused issues with backward compatibility and was reverted in 2.2] (#304)
- Clarified that implementations can, for JSON export of scopes containing no metrics, omit them, or that they can be present with an empty value. (#416)
- Clarified that metrics should not be created for private methods when a class is annotated (the TCK asserted this in 2.0 anyway) (#416)
- Added the `reusable(boolean)` method for MetadataBuilder (#407)

Specification Changes

- (2.1.1) Added ProcessCpuTime as a new optional base metric. (#454)
- Clarified that metric registry implementations are required to be thread-safe. (#300)
- Clarified that implementations can, for JSON export of scopes containing no metrics, omit them, or that they can be present with an empty value. (#416)
- Clarified that metrics should not be created for private methods when a class is annotated (the TCK asserted this in 2.0 anyway) (#416)
- Added some text to the specification about programmatic creation of metrics (without annotations) (#399)

TCK enhancement

- TCKs are updated to use RestAssured 4.0

Miscellaneous

- Explicitly excluded the transitive dependency on `jakarta.el-api` from the build of the specification. It wasn't actually used anywhere in the build so there should be no impact. Implementations can still support the Expression Language if they choose to. (#417)

Changes in 2.0

A full list of changes may be found on the MicroProfile Metrics 2.0 Milestone and MicroProfile Metrics 2.0.1 Milestone and MicroProfile Metrics 2.0.2 Milestone

Changes marked with ϟ are breaking changes relative to previous versions of the spec.

API/SPI Changes

- ϟ Refactoring of Counters, as the old `@Counted` was misleading in practice. (#290)
  - Counters via `@Counted` are now always monotonic, the `monotonic` attribute is gone. The `Counted` interface lost the `dec()` methods.
  - Former non-monotonic counters are now `@ConcurrentGauge` and also in the output reported as gauges. (#290)
  - See Migration hints about migration of applications using MicroProfile Metrics. (#290)
- Removed unnecessary `@InterceptorBinding` annotation from `org.eclipse.microprofile.metrics.annotation.Metric`. (#188)
- ϟ Removed deprecated `org.eclipse.microprofile.metrics.MetricRegistry.register(String name, Metric, Metadata)` (#268)
- ϟ `Metadata` is now immutable and built via a `MetadataBuilder`. (#228)
- Introduced a Tag object which represents a singular tag key/value pair. (#238)
- MetricFilter modified to filter with MetricID instead of name. (#238)

Functional Changes

- (2.0.3) Added ProcessCpuTime as a new optional base metric. (#454)
- ϟ `Metadata` is now immutable and built via a `MetadataBuilder`. (#228)
- ϟ Metrics are now uniquely identified by a MetricID (combination of the metric's name and tags). (#238)
- MetricFilter modified to filter with MetricID instead of name. (#238)
- The 'Metadata' is mapped to a unique metric name in the `MetricRegistry` and this relationship is immutable. (#238)
- Tag key names for labels are restricted to match the regex `[a-zA-Z_][a-zA-Z0-9_]*`. (#238)
- Tag values defined through MP_METRICS_TAGS must escape equal signs `=` and commas `,` with a backslash `\`. (#238)
- ϟ JSON output format for GET requests now appends tags along with the metric in `metricName;tag=value;tag=value` format. JSON format for OPTIONS requests have been modified such that the 'tags' attribute is a list of nested lists which holds tags from different metrics that are associated with the metadata. (#381)
- OpenMetrics format - formerly called Prometheus format

    - Reserved characters in OpenMetrics format must be escaped. (#238)
    - ϟ In OpenMetrics output format, the separator between scope and metric name is now a `_` instead of a `:`. (#279)
    - ϟ Metric names with camelCase are no longer converted to snake_case for OpenMetrics output. (#357)
- ϟ The default value of the `reusable` attribute for metric objects created programmatically (not via annotations) is now `true` (#328)
- ϟ Some base metrics' names have changed to follow the convention of ending the name of accumulating counters with `total`. (#375)
- ϟ Some base metrics' types have changed from Counter to Gauge since Counters must now count monotonically. (#375)
- ϟ Some base metrics' names have changed because they now use tags to distinguish metrics for multiple JVM objects. For example, each existing garbage collector now has its own `gc.total` metric with the name of the garbage collector being in a tag. Names of some base metrics in the OpenMetrics output are also affected by the removal of conversion from camelCase to snake_case. (#375)

Specification Changes

- (2.0.3) Added ProcessCpuTime as a new optional base metric. (#454)
- ϟ Refactoring of Counters, as the old `@Counted` was misleading in practice. (#290)
    - Counters via `@Counted` are now always monotonic, the `monotonic` attribute is gone. The `Counted` interface lost the `dec()` methods.
    - Former non-monotonic counters are now `@ConcurrentGauge` and also in the output reported as gauges. (#290)
    - See Migration hints about migration of applications using MicroProfile Metrics. (#290)
- ϟ Metrics are now uniquely identified by a MetricID (combination of the metric's name and tags). (#238)
- The 'Metadata' is mapped to a unique metric name in the `MetricRegistry` and this relationship is immutable. (#238)
- Tag key names for labels are restricted to match the regex `[a-zA-Z_][a-zA-Z0-9_]*`. (#238)
- Tag values defined through MP_METRICS_TAGS must escape equal signs `=` and commas `,` with a backslash `\`. (#238)
- OpenMetrics format - formerly called Prometheus format
    - Reserved characters in OpenMetrics format must be escaped. (#238)
    - ϟ In OpenMetrics output format, the separator between scope and metric name is now a `_` instead of a `:`. (#279)
    - ϟ Metric names with camelCase are no longer converted to snake_case for OpenMetrics output. (#357)
- ϟ The default value of the `reusable` attribute for metric objects created programmatically (not

via annotations) is now `true` (#328)

- ϟ Some base metrics' names have changed to follow the convention of ending the name of accumulating counters with `total`. (#375)
- ϟ Some base metrics' types have changed from Counter to Gauge since Counters must now count monotonically. (#375)
- ϟ Some base metrics' names have changed because they now use tags to distinguish metrics for multiple JVM objects. For example, each existing garbage collector now has its own `gc.total` metric with the name of the garbage collector being in a tag. Names of some base metrics in the OpenMetrics output are also affected by the removal of conversion from camelCase to snake_case. (#375)
- Added a set of recommendations how application servers with multiple deployed applications should behave if they support MP Metrics. (#240)

Changes in 1.1

A full list of changes may be found on the MicroProfile Metrics 1.1 Milestone

API/SPI Changes

- `org.eclipse.microprofile.metrics.MetricRegistry.register(String name, Metric, Metadata)` is deprecated. Use `org.eclipse.microprofile.metrics.MetricRegistry.register(Metadata, Metric)` instead, where `Metadata` already has a field for the name.

Functional Changes

- `org.eclipse.microprofile.metrics.MetricRegistry.register(String name, Metric, Metadata)` is deprecated. Use `org.eclipse.microprofile.metrics.MetricRegistry.register(Metadata, Metric)` instead, where `Metadata` already has a field for the name.
- Global tags are now supplied via the means of MicroProfile Config (the env variable is still valid). (#165)

Specification Changes

- Annotations and `Metadata` can now have a flag `reusable` that indicates that the metric name can be registered more than once. Default is `false` as in Metrics 1.0. See Reusing Metrics.

TCK enhancement

- Improved TCK