Metrics for Eclipse MicroProfile

Heiko W. Rupp, Raymond Lam, David Chan, Don Bourne, Antonin Stefanutti,
Brennan Nichyporuk, Mike Croft, Werner Keil, Jan Martiska

Version 2.3-M5, January 22, 2020

Table of Contents

Chapter 1. MicroProfile Metrics

```
Specification: Metrics for Eclipse MicroProfile

Version: 2.3-M5

Status: Draft

Release: January 22, 2020

Copyright (c) 2016-2019 Eclipse Microprofile Contributors:
Heiko W. Rupp, Raymond Lam, David Chan, Don Bourne, Antonin Stefanutti, Brennan
Nichyporuk, Mike Croft, Werner Keil, Jan Martiska

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

   http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

Chapter 2. 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.

2.1. 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.

2.2. 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.

Chapter 3. 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 Required Metrics for more information regarding metrics that are required for each vendor.

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

3.1. 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: metrics that all MicroProfile vendors have to provide
- vendor: vendor specific metrics (optional)
- application: application-specific metrics (optional)

It is expected that a future version of this specification will also have a sub-resource for integrations with other specifications of MicroProfile.

3.1.1. Scopes

Required Base metrics

Required base metrics describe a set of metrics that all MicroProfile-compliant servers have to provide. Each vendor can implement the set-up of the metrics in the *base* scope 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 Required Metrics lists the required metrics. This list also includes a few items marked as optional. These are listed here as they are dependent on the underlying JVM and not the server and thus fit better in *base* scope than the *vendor* one.

Required base metrics are exposed under `/metrics/base`.

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 implementations of the MicroProfile. 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.

Application specific metrics are exposed under `/metrics/application`.

Vendor specific Metrics

It is possible for MicroProfile server implementors to supply their specific metrics data on top of the basic set of required metrics. Vendor specific metrics are exposed under `/metrics/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

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.

3.1.2. Tags

Tags (or 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 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 1.3 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.

In application servers with multiple applications deployed, there is one reserved tag name: `_app`, which serves for distinguishing 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.

3.1.3. 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
- type:
  - counter: a monotonically increasing numeric value (e.g. total number of requests received).
  - concurrent gauge: an incrementally increasing or decreasing numeric value (e.g. number of parallel invocations of a method).

This type exposes three values: current count, highest count within the previous full minute and lowest count within the previous full minute.

Full minute is the minute from second 0 to just before second 0 on the next minute (eg. from [10:46:00-10:46:59.99999999]).

  - gauge: a metric that is sampled to obtain its value (e.g. cpu temperature or disk usage).
  - meter: a metric which tracks mean throughput and one-, five-, and fifteen-minute exponentially-weighted moving average throughput.
  - histogram: a metric which calculates the distribution of a value.
  - timer: a metric which aggregates timing durations and provides duration statistics, plus throughput statistics.
  - simple timer: a lightweight alternative to the timer metric that only tracks the elapsed time duration and invocation counts. The simple timer may be preferrable over the timer when used with Prometheus as the statistical calculations can be deferred to Prometheus using the simple timer's available values.

- description (optional): A human readable description of the metric.
- displayName (optional): A human readable name of the metric for display purposes if the metric name is not human readable. This could e.g. be the case when the metric name is a uuid.
- reusable (optional): If set to `true`, then it is allowed to register a metric multiple times under the same MetricID. Note that all such instances must set `reusable` to `true`. Default is `true` for metrics created programmatically, `false` for metrics declared using annotations. See Reusing Metrics for more details.

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.

3.2. Metric Registry

The `MetricRegistry` stores the metrics and metadata information. There is one `MetricRegistry` instance for each of the 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 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.

3.2.1. 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.

3.2.2. Reusing Metrics

For metrics declared using annotations, by default it is not allowed to register more than one metric under a certain name and tags combination in a scope. This is done to prevent hard to spot copy & paste errors, where for example all methods of a JAX-RS class are marked with `@Timed(name="myApp", absolute=true)`.

If this behaviour is required, then it is possible to mark all such instances as *reusable* by passing the respective flag in the Annotation. Gauges are not reusable.

For metrics created programmatically (by calling methods of the `MetricRegistry`), reusing is allowed by default, so multiple calls retrieving an instance of a metric from the registry will return the same metric object so that the object can be reused in multiple places in the application.

The implementation must throw an 'IllegalArgumentException' during a metric registration call when the call would result in the reuse of a metric where that metric was either previously declared not reusable or where the registration call itself declares the metric to not be reusable.

Only metrics of the same type can be reused under the same MetricID. Trying to reuse a name for different types will result in an `IllegalArgumentException`. All metrics under the same name must also have exactly the same metadata.

If you want to re-use a MetricID, then you need to also explicitly set the `name` field OR set `absolute` to `true` and have multiple methods annotated as metric that have the same method name and tags.

Example of reused counters

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

@Counted(name = "countMe", absolute = true, reusable = 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.

3.2.3. 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. Concurrent gauges will watch the number of parallel invocations of a method even if the invocations are on different instances.

The only exception from this are gauges (not concurrent 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.

3.3. Exposing metrics via REST API

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

- JSON format - used when the HTTP Accept header best matches `application/json`.
- OpenMetrics text format - used when the HTTP Accept header best matches `text/plain` or when Accept header would equally accept both `text/plain` and `application/json` and there is no other higher precedence format. 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 OpenMetrics text 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	JSON, OpenMetrics	Returns all registered metrics
`/metrics/<scope>`	GET	JSON, OpenMetrics	Returns metrics registered for the respective scope. Scopes are listed in Metrics Setup
`/metrics/<scope>/<metric_name>`	GET	JSON, OpenMetrics	Returns the metric that matches the metric name for the respective scope
`/metrics`	OPTIONS	JSON	Returns all registered metrics' metadata
`/metrics/<scope>`	OPTIONS	JSON	Returns metrics' metadata registered for the respective scope. Scopes are listed in Metrics Setup
`/metrics/<scope>/<metric_name>`	OPTIONS	JSON	Returns the metric's metadata that matches the metric name for the respective scope

The implementation must return a 406 response code if the request's HTTP Accept header for an OPTIONS request does not match `application/json`.

3.4. 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 `_app` should be appended to each metric. Its value should be equal to the context root of the web application to which the metric belongs. For example, if a deployment is available under the `/cars` context root, each metric created by this deployment will contain an additional tag named `_app` with a value of `/cars`. If the application server allows using metrics in JAR deployments, which have no web context, the name of the JAR archive (including the `.jar` suffix) should be used. If such JAR is a module of an EAR application, the value of the `_app` tag should be `ear_name#jar_name`.

This is an example JSON output from an application server that has applications under `/app1` and `/app2`, both of which have a counter metric named `requestCount`:

```
{
  "requestCount;_app=/app1" : 198,
  "requestCount;_app=/app2" : 320
}
```

The value of the `_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 `_app` tag at all, but in that case, metrics from two applications on one server can clash as no differentiator (by application) is given.

There should be a single `MetricRegistry` instance shared between all applications to prevent unexpected clashes when merging the contents of different registries while responding to metric export requests. It is up to the application server whether it will allow sharing of metrics between different applications (for example, if there's a reusable metric in one application, another might want to reuse it).

3.4.1. Implementation notes:

Constructors of the `MetricID` class from the API code already handle adding the `_app` tag automatically when they detect that there is a property named `mp.metrics.appName` available from the `org.eclipse.microprofile.config.Config` instance available in the current context. If no such property exists or if the value is empty, no tag will be appended.

Generally, the responsibility of the application server implementation will be to append a property `mp.metrics.appName` to the `org.eclipse.microprofile.config.Config` instance of each application during deployment time, its value being the web context root of the application or the JAR name. This can be achieved for example by adding a custom `ConfigSource` with an ordinal less than 100, because the `ConfigSource` that reads properties `META-INF/microprofile-config.properties` has an ordinal of 100, and this needs to have higher priority.

Chapter 4. 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

4.1. JSON format

- When using JSON format, the REST API will respond to GET requests with data formatted in a tree like fashion with sub-trees for the sub-resources. A sub-tree that does not contain data must be omitted.
- A 'shadow tree' that responds to OPTIONS will provide the metadata and tags associated to a metric name.

4.1.1. Translation rules for metric names and handling of tags

The following rules apply only to GET requests:

- Tags are appended to the leaf element of the metric's JSON tree.
- For metrics with tags, the metric name must be appended with a semicolon `;` followed by a semicolon-separated list of tag key/value pairs.
- For compound metrics (those with child JSON attributes) with tags, only the "leaf" metric names are decorated with tags.
- Semicolons `;` present in tag values must be converted to underscores `_` in JSON output.

For example:

```
{
 "carsCounter;colour=red" : 0,
 "carsCounter;colour=blue;car=sedan" : 0,
 "carsMeter": {
    "count;colour=red" : 0,
    "meanRate;colour=red" : 0,
    "oneMinRate;colour=red" : 0,
    "fiveMinRate;colour=red" : 0,
    "fifteenMinRate;colour=red" : 0,
    "count;colour=blue" : 0,
    "meanRate;colour=blue" : 0,
    "oneMinRate;colour=blue" : 0,
    "fiveMinRate;colour=blue" : 0,
    "fifteenMinRate;colour=blue" : 0
 }
}
```

The following apply to both GET and OPTION requests:

- Each tag is a key-value-pair in the format of `<key>=<value>`. The list of tags must be sorted alphabetically by key name.
- If the metric name or tag value contains a special reserved JSON character, these characters must be escaped in the JSON response.

If the metric has no tags, the semicolon `;` must be omitted.

For example,

```
{
  "metricWithoutTags": 192
}
```

REST-API Objects

API-objects MAY include one or more metrics as in

```
{
  "thread.count" : 33,
  "thread.max.count" : 47,
  "memory.maxHeap" : 3817863211,
  "memory.usedHeap" : 16859081,
  "memory.committedHeap" : 64703546
}
```

or

```
{
  "hitCount;type=yes": 45
}
```

In case `/metrics` is requested, then the data for the scopes are wrapped in the scope name:

```
{
  "application": {
    "hitCount": 45
  },
  "base": {
     "thread.count" : 33,
     "thread.max.count" : 47
  },
  "vendor": {...}
}
```

If there is a scope that contains no metrics, then it can be either present with an empty object as its value, or it can be omitted completely.

4.1.2. Gauge JSON Format

The value of the gauge must be equivalent to a call to the instance Gauge's `getValue()`.

Example Gauge JSON GET Response

```
{
  "responsePercentage": 48.45632,
  "responsePercentage;servlet=two": 26.23654,
  "responsePercentage;store=webshop;servlet=three": 29.24554
}
```

4.1.3. Counter JSON Format

The value of the counter must be equivalent to a call to the instance Counter's `getCount()`.

Example Counter JSON GET Response

```
{
  "hitCount": 45,
  "hitCount;servlet=two": 3,
  "hitCount;store=webshop;servlet=three": 4
}
```

4.1.4. Concurrent Gauge JSON Format

`ConcurrentGauge` is a complex metric type comprised of multiple key/values. The format is specified by the table below.

Table 2. JSON mapping for a ConcurrentGauge metric

JSON Key	Value (Equivalent ConcurrentGauge method)
`current`	`getValue()`
`min`	`getMin()`
`max`	`getMax()`

Example ConcurrentGauge JSON GET Response

```
{
  "callCount": {
      "current" : 48,
      "min": 4,
      "max": 50,
      "current;component=backend" : 23,
      "min;component=backend": 1,
      "max;component=backend": 29
  }
}
```

4.1.5. Meter JSON Format

`Meter` is a complex metric type comprised of multiple key/values. The format is specified by the table below.

Table 3. JSON mapping for a Meter metric

JSON Key	Value (Equivalent Meter method)
`count`	`getCount()`
`meanRate`	`getMeanRate()`
`oneMinRate`	`getOneMinuteRate()`
`fiveMinRate`	`getFiveMinuteRate()`
`fifteenMinRate`	`getFifteenMinuteRate()`

Example Meter JSON GET Response

```
{
  "requests": {
    "count": 29382,
    "meanRate": 12.223,
    "oneMinRate": 12.563,
    "fiveMinRate": 12.364,
    "fifteenMinRate": 12.126,
    "count;servlet=one": 29382,
    "meanRate;servlet=one": 12.223,
    "oneMinRate;servlet=one": 12.563,
    "fiveMinRate;servlet=one": 12.364,
    "fifteenMinRate;servlet=one": 12.126,
    "count;servlet=two": 29382,
    "meanRate;servlet=two": 12.223,
    "oneMinRate;servlet=two": 12.563,
    "fiveMinRate;servlet=two": 12.364,
    "fifteenMinRate;servlet=two": 12.126
  }
}
```

4.1.6. Histogram JSON Format

`Histogram` is a complex metric type comprised of multiple key/values. The format is specified by the table below.

Table 4. JSON mapping for a Histogram metric

JSON Key	Value (Equivalent Histogram method)
`count`	`getCount()`
`min`	`getSnapshot().getMin()`
`max`	`getSnapshot().getMax()`
`mean`	`getSnapshot().getMean()`
`stddev`	`getSnapshot().getStdDev()`

JSON Key	Value (Equivalent Histogram method)
p50	getSnapshot().getMedian()
p75	getSnapshot().get75thPercentile()
p95	getSnapshot().get95thPercentile()
p98	getSnapshot().get98thPercentile()
p99	getSnapshot().get99thPercentile()
p999	getSnapshot().get999thPercentile()

Example Histogram JSON GET Response


```
{
  "daily_value_changes": {
    "count":2,
    "min":-1624,
    "max":26,
    "mean":-799.0,
    "stddev":825.0,
    "p50":26.0,
    "p75":26.0,
    "p95":26.0,
    "p98":26.0,
    "p99":26.0,
    "p999":26.0,
    "count;servlet=two":2,
    "min;servlet=two":-1624,
    "max;servlet=two":26,
    "mean;servlet=two":-799.0,
    "stddev;servlet=two":825.0,
    "p50;servlet=two":26.0,
    "p75;servlet=two":26.0,
    "p95;servlet=two":26.0,
    "p98;servlet=two":26.0,
    "p99;servlet=two":26.0,
    "p999;servlet=two":26.0
  }
}
```


4.1.7. Timer JSON Format

`Timer` is a complex metric type comprised of multiple key/values. The format is specified by the table below.

Table 5. JSON mapping for a Timer metric

JSON Key	Value (Equivalent Timer method)
count	getCount()
meanRate	getMeanRate()
oneMinRate	getOneMinuteRate()

JSON Key	Value (Equivalent Timer method)
fiveMinRate	getFiveMinuteRate()
fifteenMinRate	getFifteenMinuteRate()
min	getSnapshot().getMin()
max	getSnapshot().getMax()
mean	getSnapshot().getMean()
stddev	getSnapshot().getStdDev()
p50	getSnapshot().getMedian()
p75	getSnapshot().get75thPercentile()
p95	getSnapshot().get95thPercentile()
p98	getSnapshot().get98thPercentile()
p99	getSnapshot().get99thPercentile()
p999	getSnapshot().get999thPercentile()

Example Timer JSON GET Response


```
{
  "responseTime": {
    "count": 29382,
    "meanRate":12.185627192860734,
    "oneMinRate": 12.563,
    "fiveMinRate": 12.364,
    "fifteenMinRate": 12.126,
    "min":169916,
    "max":5608694,
    "mean":415041.00024926325,
    "stddev":652907.9633011606,
    "p50":293324.0,
    "p75":344914.0,
    "p95":543647.0,
    "p98":2706543.0,
    "p99":5608694.0,
    "p999":5608694.0,
    "count;servlet=two": 29382,
    "meanRate;servlet=two":12.185627192860734,
    "oneMinRate;servlet=two": 12.563,
    "fiveMinRate;servlet=two": 12.364,
    "fifteenMinRate;servlet=two": 12.126,
    "min;servlet=two":169916,
    "max;servlet=two":5608694,
    "mean;servlet=two":415041.00024926325,
    "stddev;servlet=two":652907.9633011606,
    "p50;servlet=two":293324.0,
    "p75;servlet=two":344914.0,
    "p95;servlet=two":543647.0,
    "p98;servlet=two":2706543.0,
    "p99;servlet=two":5608694.0,
    "p999;servlet=two":5608694.0
  }
}
```


4.1.8. Simple Timer JSON Format

`Simple Timer` is a complex metric type comprised of multiple key/values. The format is specified by the table below.

Table 6. JSON mapping for a Simple Timer metric

JSON Key	Value (Equivalent SimpleTimer method)
`count`	`getCount()`
`elapsedTime`	`getElapsedTime()`

Example Simple Timer JSON GET Response

```
{
  "simple_responseTime": {
    "count": 1,
    "elapsedTime": 12300000000
  }
}
```

4.1.9. Metadata

Metadata is exposed in a tree-like fashion with sub-trees for the sub-resources mentioned previously. Tags from metrics associated with the metric name are also included. The 'tags' attribute is an array of nested arrays which hold tags from different metrics that are associated with the metadata.

Example:

If `GET /metrics/base/fooVal` exposes:

```
{
  "fooVal;store=webshop": 12345
}
```

then `OPTIONS /metrics/base/fooVal` will expose:

```
{
  "fooVal": {
    "unit": "milliseconds",
    "type": "gauge",
    "description": "The size of foo after each request",
    "displayName": "Size of foo",
    "tags": [
      [
        "store=webshop"
      ]
    ]
  }
}
```

If `GET /metrics/base` exposes multiple values like this:

Example of exposed metrics data

```
{
  "fooVal;store=webshop": 12345,
  "barVal;store=webshop;component=backend": 42,
  "barVal;store=webshop;component=frontend": 63
}
```

then `OPTIONS /metrics/base` exposes:

Example of JSON output of Metadata

```
{
  "fooVal": {
    "unit": "milliseconds",
    "type": "gauge",
    "description": "The average duration of foo requests during last 5 minutes",
    "displayName": "Duration of foo",
    "tags": [
      [
        "store=webshop"
      ]
    ]
  },
  "barVal": {
    "unit": "megabytes",
    "type": "gauge",
    "tags": [
      [
        "store=webshop",
        "component=backend"
      ],
      [
        "store=webshop",
        "component=frontend"
      ]
    ]
  }
}
```

4.2. OpenMetrics format

Data is exposed in the OpenMetrics text format, version 0.0.4 as described in OpenMetrics text format.

The metadata will be included as part of the normal OpenMetrics text format. Unlike the JSON format, the text format does not support OPTIONS requests.

Users that want to write tools to transform the metadata can still request the metadata via OPTIONS request and `application/json` media type.

The above json example would look like this in OpenMetrics format

Example of OpenMetrics output

```
# TYPE base_fooVal_seconds gauge
# HELP base_fooVal_seconds The average duration of foo requests during last 5 minutes
①
base_fooVal_seconds{store="webshop"} 12.345  ②
# TYPE base_barVal_bytes gauge
base_barVal_bytes{component="backend", store="webshop"} 42000 ②
# TYPE base_barVal_bytes gauge
base_barVal_bytes{component="frontend", store="webshop"} 63000 ②
```

① The description goes into the HELP line

② Metric names gets the base unit of the family appended with `_` and defined labels. Values are scaled accordingly. See Handling of units

4.2.1. Translation rules for metric names

OpenMetrics text format does not allow for all characters and adds the base unit of a family to the name. Characters allowed are `[a-zA-Z0-9_]` (Ascii alphabet, numbers and underscore). Exposed metric names must follow the pattern `[a-zA-Z_][a-zA-Z0-9_]*`.

- Characters that do not fall in above category are translated to underscore (_).
- Scope is always specified at the start of the metric name.
- Scope and name are separated by underscore (_).
- Double underscore is translated to single underscore
- The unit is appended to the name, separated by underscore. See Handling of units

4.2.2. Handling of tags

Metric tags are appended to the metric name in curly braces `{` and `}` and are separated by comma. Each tag is a key-value-pair in the format of `<key>="<value>"` (the quotes around the value are required).

MicroProfile Metrics timers and histograms expose an OpenMetrics `summary` type which requires an additional `quantile` tag for certain metrics. The `quantile` tag must be included alongside the metrics tags within the curly braces `{` and `}`.

The tag value can be any Unicode character but the following characters must be escaped:

- Backslash (`\`) must be escaped as `\\` (as two characters: `\` and `\`)
- Double-quotes (`"`) must be escaped as `\"` (as two characters: `\` and `"`)
- Line feed (`\n`) must be escaped as `\n` (as two characters: `\` and `n`)

4.2.3. Handling of units

The OpenMetrics text format adheres to using "base units" when creating the HTTP response. Due to the different context of each metric type, certain metrics' values must be converted to the respective "base unit" when responding to OpenMetrics requests. For example, times in milliseconds must be divided by 1000 and displayed in the base unit (seconds).

The following sections outline how each metric type is handled:

Gauges and Histograms

The metric name and values for `Gauge` and `Histogram` are converted to the "base unit" in respect to the `unit` value in the Metadata.

- If the Metadata is empty, `NONE`, or null, the metric name is used as is without appending the unit name and no scaling is applied.
- If the metric's metadata contains a known unit, as defined in the `MetricUnits` class, the OpenMetrics value should be scaled to the *base unit* of the respective family. The name of the base unit is appended to the metric name delimited by underscores (_).
- If the `unit` is specified and is not defined in the `MetricUnits` class, the value is not scaled but the `unit` is still appended to the metric name delimited by underscores (_).

Unit families and their base units are described under OpenMetrics metric names, Base units.

Families and OpenMetrics base units are:

Family	Base unit
Bits	bytes
Bytes	bytes
Time	seconds
Percent	ratio (normally ratio is A_per_B, but there are exceptions like `disk_usage_ratio`)

Counters

`Counter` metrics are considered dimensionless. The implementation must not append the unit name to the metric name and must not scale the value.

Meters and Timers

`Meter` and `Timer` have fixed units as described below regardless of the `unit` value in the Metadata.

4.2.4. Gauge OpenMetrics Text Format

The value of the gauge must be the value of `getValue()` with appropriate naming/scaling based on Handling of units

Example OpenMetrics text format for a Gauge in dollars.

```
# TYPE application_cost_dollars gauge
# HELP application_cost_dollars The running cost of the server in dollars.
application_cost_dollars 80
```

4.2.5. Counter OpenMetrics Text Format

The value of the counter must be the value of `getCount()`. The exposed metric name must have a `_total` suffix. The suffix is not appended if the (translated) original metric name already ends in `_total`. Counters do not have a suffix for the unit.

Example OpenMetrics text format for a Counter.

```
# TYPE application_visitors_total counter
# HELP application_visitors_total The number of unique visitors
application_visitors_total 80
```

4.2.6. Concurrent Gauge OpenMetrics Text Format

`ConcurrentGauge` is a complex metric type comprised of multiple key/values. Each key will require a suffix to be appended to the metric name. The format is specified by the table below.

Table 7. OpenMetrics text mapping for a ConcurrentGauge metric

Suffix{label}	TYPE	Value (Meter method)	Units
`current`	Gauge	`getCount()`	N/A
`min`	Gauge	`getMin()`	N/A
`max`	Gauge	`getMax()`	N/A

Concurrent gauges do not have a suffix for the unit.

Example OpenMetrics text format for a Concurrent Gauge

```
# TYPE application_method_a_invocations_current gauge
# HELP application_method_a_invocations_current The number of parallel invocations of
methodA() ①
application_method_a_invocations_current 80
# TYPE application_method_a_invocations_min gauge
application_method_a_invocations_min 20
# TYPE application_method_a_invocations_max gauge
application_method_a_invocations_max 100
```

① Note help is only emitted for the metric related to `getCount()`, but not for _min and _max.

4.2.7. Meter OpenMetrics Text Format

`Meter` is a complex metric type comprised of multiple key/values. Each key will require a suffix to be

appended to the metric name. The format is specified by the table below.

The `# HELP` description line is only required for the `total` value as shown below.

Table 8. OpenMetrics text mapping for a Meter metric

Suffix{label}	TYPE	Value (Meter method)	Units
`total`	Counter	`getCount()`	N/A
`rate_per_second`	Gauge	`getMeanRate()`	PER_SECOND
`one_min_rate_per_second`	Gauge	`getOneMinuteRate()`	PER_SECOND
`five_min_rate_per_second`	Gauge	`getFiveMinuteRate()`	PER_SECOND
`fifteen_min_rate_per_second`	Gauge	`getFifteenMinuteRate()`	PER_SECOND

Example OpenMetrics text format for a Meter

```
# TYPE application_requests_total counter
# HELP application_requests_total Tracks the number of requests to the server
application_requests_total 29382
# TYPE application_requests_rate_per_second gauge
application_requests_rate_per_second 12.223
# TYPE application_requests_one_min_rate_per_second gauge
application_requests_one_min_rate_per_second 12.563
# TYPE application_requests_five_min_rate_per_second gauge
application_requests_five_min_rate_per_second 12.364
# TYPE application_requests_fifteen_min_rate_per_second gauge
application_requests_fifteen_min_rate_per_second 12.126
```

4.2.8. Histogram OpenMetrics Text Format

`Histogram` is a complex metric type comprised of multiple key/values. Each key will require a suffix to be appended to the metric name with appropriate naming/scaling based on Handling of units. The format is specified by the table below.

The `# HELP` description line is only required for the `summary` value as shown below.

Table 9. OpenMetrics text mapping for a Histogram metric

Suffix{label}	TYPE	Value (Histogram method)	Units
`min_<units>`	Gauge	`getSnapshot().getMin()`	<units>[1]
`max_<units>`	Gauge	`getSnapshot().getMax()`	<units>[1]
`mean_<units>`	Gauge	`getSnapshot().getMean()`	<units>[1]
`stddev_<units>`	Gauge	`getSnapshot().getStdDev()`	<units>[1]
`<units>_count`[2]	Summary	`getCount()`	N/A
`<units>{quantile="0.5"}`[2]	Summary	`getSnapshot().getMedian()`	<units>[1]
`<units>{quantile="0.75"}`[2]	Summary	`getSnapshot().get75thPercentile()`	<units>[1]

Suffix{label}	TYPE	Value (Histogram method)	Units
<units>{quantile="0.95"}[2]	Summary	getSnapshot().get95thPercentile()	<units>[1]
<units>{quantile="0.98"}[2]	Summary	getSnapshot().get98thPercentile()	<units>[1]
<units>{quantile="0.99"}[2]	Summary	getSnapshot().get99thPercentile()	<units>[1]
<units>{quantile="0.999"}[2]	Summary	getSnapshot().get999thPercentile()	<units>[1]

[1] The implementation is expected to convert the result returned by the `Histogram` into the base unit (if known). The `<unit>` represents the base metric unit and is named according to Handling of units.

[2] The `summary` type is a complex metric type for OpenMetrics which consists of the count and multiple quantile values.

Example OpenMetrics text format for a Histogram with unit bytes.

```
# TYPE application_file_sizes_mean_bytes gauge
application_file_sizes_mean_bytes 4738.231
# TYPE application_file_sizes_max_bytes gauge
application_file_sizes_max_bytes 31716
# TYPE application_file_sizes_min_bytes gauge
application_file_sizes_min_bytes 180
# TYPE application_file_sizes_stddev_bytes gauge
application_file_sizes_stddev_bytes 1054.7343037063602
# TYPE application_file_sizes_bytes summary
# HELP application_file_sizes_bytes Users file size
application_file_sizes_bytes_count 2037
application_file_sizes_bytes{quantile="0.5"} 4201
application_file_sizes_bytes{quantile="0.75"} 6175
application_file_sizes_bytes{quantile="0.95"} 13560
application_file_sizes_bytes{quantile="0.98"} 29643
application_file_sizes_bytes{quantile="0.99"} 31716
application_file_sizes_bytes{quantile="0.999"} 31716
```

4.2.9. Timer OpenMetrics Text Format

`Timer` is a complex metric type comprised of multiple key/values. Each key will require a suffix to be appended to the metric name. The format is specified by the table below.

The `# HELP` description line is only required for the `summary` value as shown below.

Table 10. OpenMetrics text mapping for a Timer metric

Suffix{label}	TYPE	Value (Timer method)	Units
rate_per_second	Gauge	getMeanRate()	PER_SECOND
one_min_rate_per_second	Gauge	getOneMinuteRate()	PER_SECOND
five_min_rate_per_second	Gauge	getFiveMinuteRate()	PER_SECOND

Suffix{label}	TYPE	Value (Timer method)	Units
fifteen_min_rate_per_second	Gauge	getFifteenMinuteRate()	PER_SECOND
min_seconds	Gauge	getSnapshot().getMin()	SECONDS[1]
max_seconds	Gauge	getSnapshot().getMax()	SECONDS[1]
mean_seconds	Gauge	getSnapshot().getMean()	SECONDS[1]
stddev_seconds	Gauge	getSnapshot().getStdDev()	SECONDS[1]
seconds_count[2]	Summary	getCount()	N/A
seconds{quantile="0.5"}[2]	Summary	getSnapshot().getMedian()	SECONDS[1]
seconds{quantile="0.75"}[2]	Summary	getSnapshot().get75thPercentile()	SECONDS[1]
seconds{quantile="0.95"}[2]	Summary	getSnapshot().get95thPercentile()	SECONDS[1]
seconds{quantile="0.98"}[2]	Summary	getSnapshot().get98thPercentile()	SECONDS[1]
seconds{quantile="0.99"}[2]	Summary	getSnapshot().get99thPercentile()	SECONDS[1]
seconds{quantile="0.999"}[2]	Summary	getSnapshot().get999thPercentile()	SECONDS[1]

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

[2] The `summary` type is a complex metric type for OpenMetrics which consists of the count and multiple quantile values.

Example OpenMetrics text format for a Timer

```
# TYPE application_response_time_rate_per_second gauge
application_response_time_rate_per_second  0.004292520715985437
# TYPE application_response_time_one_min_rate_per_second gauge
application_response_time_one_min_rate_per_second  2.794076465421066E-14
# TYPE application_response_time_five_min_rate_per_second  gauge
application_response_time_five_min_rate_per_second  4.800392614619373E-4
# TYPE application_response_time_fifteen_min_rate_per_second  gauge
application_response_time_fifteen_min_rate_per_second  0.01063191047532505
# TYPE application_response_time_mean_seconds gauge
application_response_time_mean_seconds 0.000415041
# TYPE application_response_time_max_seconds gauge
application_response_time_max_seconds 0.0005608694
# TYPE application_response_time_min_seconds gauge
application_response_time_min_seconds 0.000169916
# TYPE application_response_time_stddev_seconds gauge
application_response_time_stddev_seconds 0.000652907
# TYPE application_response_time_seconds summary
# HELP application_response_time_seconds Server response time for /index.html
application_response_time_seconds_count 80
application_response_time_seconds{quantile="0.5"} 0.0002933240
application_response_time_seconds{quantile="0.75"} 0.000344914
application_response_time_seconds{quantile="0.95"} 0.000543647
application_response_time_seconds{quantile="0.98"} 0.002706543
application_response_time_seconds{quantile="0.99"} 0.005608694
application_response_time_seconds{quantile="0.999"} 0.005608694
```

4.2.10. Simple Timer OpenMetrics Text Format

`Simple Timer` is a complex metric type comprised of multiple key/values. Each key will require a suffix to be appended to the metric name. The format is specified by the table below.

Table 11. OpenMetrics text mapping for a SimpleTimer metric

Suffix{label}	TYPE	Value (SimpleTimer method)	Units
`total`	Counter	`getCount()`	N/A
`elapsedTime_seconds`	Gauge	`getElapsedTime()`	SECONDS[1]

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

Example OpenMetrics text format for a SimpleTimer

```
# TYPE application_response_time_total counter
# HELP application_response_time_total The number of calls to this REST endpoint #(1)
application_response_time_total 12
# TYPE application_response_time_elapsedTime_seconds gauge
application_response_time_elapsedTime_seconds  12.3
```

① Note help is only emitted for the metric related to `getCount()`, but not for elapsedTime.

4.3. 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)

Chapter 5. Required Metrics

Base metrics is a list of metrics that all vendors need to implement. Optional base metrics are recommended to be implemented but are not required. These metrics are exposed under `/metrics/base`.

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

Although vendors are required to implement these base metrics, some virtual machines can not provide them. Vendors should either use other metrics that are close enough as substitute or not fill these base metrics at all.

5.1. General JVM Stats

UsedHeapMemory

Name	memory.usedHeap
DisplayName	Used Heap Memory
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
DisplayName	Committed Heap Memory
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
DisplayName	Max Heap Memory
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
DisplayName	Garbage Collection Count
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
DisplayName	Garbage Collection Time
Type	Gauge
Unit	Milliseconds
Multi	true
Tags	{name=%s}
Description	Displays the approximate accumulated collection elapsed time in milliseconds. 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.

JVM Uptime - Up time of the Java Virtual machine

Name	jvm.uptime
DisplayName	JVM Uptime
Type	Gauge
Unit	Milliseconds
Description	Displays the time elapsed since the start of the Java virtual machine in milliseconds.
MBean	java.lang:type=Runtime/Uptime
Notes	Also from JSR 77

5.2. Thread JVM Stats

ThreadCount

Name	thread.count
DisplayName	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
DisplayName	Daemon Thread 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
DisplayName	Peak Thread 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

5.3. Thread Pool Stats

(Optional) ActiveThreads

Name	threadpool.activeThreads
DisplayName	Active Threads
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.

(Optional) PoolSize

Name	threadpool.size
DisplayName	Thread Pool 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.

5.4. ClassLoading JVM Stats

LoadedClassCount

Name	classloader.loadedClasses.count
DisplayName	Current Loaded Class 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
DisplayName	Total Loaded Class Count

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
DisplayName	Total Unloaded Class Count
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

5.5. Operating System

AvailableProcessors

Name	cpu.availableProcessors
DisplayName	Available Processors
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

(Optional) SystemLoadAverage

Name	cpu.systemLoadAverage
DisplayName	System Load Average
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

(Optional) ProcessCpuLoad

Name	cpu.processCpuLoad
DisplayName	Process CPU Load
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

(Optional) ProcessCpuTime

Name	cpu.processCpuTime
DisplayName	Process CPU Time
Type	Gauge
Unit	Nanoseconds
Description	Displays the CPU time used by the process on which the Java virtual machine is running in nanoseconds.
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

Chapter 6. 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: The programming API was inspired by Dropwizard Metrics 3.2.3 API, with some changes. It is expected that many existing DropWizard Metrics based applications can easily be ported over by switching the package names.
- 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.

6.1. 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.
- 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 flag duplicate metrics upon registration and reject the duplicate unless the metric is explicitly marked as reusable upon first registration and in all subsequent registrations.
  - A duplicate metric is a metric that has the same scope and MetricID (name and tags) as an existing one.
  - The implementation must throw an `IllegalArgumentException` when the metric is rejected.
  - It is not allowed to reuse a metric (name) for metrics of different types. The implementation must throw an `IllegalArgumentException` if such a mismatch is detected.
  - See reusing of metrics for more details.
- The implementation must flag and 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 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, @SimplyTimed, @Counted, @ConcurrentGauge, @Metered
- 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.

6.2. Base Package

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

The `org.eclipse.microprofile.metrics` package was influenced by the Drop Wizard Metrics project release 3.2.3.

Implementors can consult this project for implementation ideas.

See References for more information.

6.3. 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 on the Java Interceptors specification to provide the ability to associate interceptors to beans via typesafe interceptor bindings, as a mean 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 `org.eclipse.microprofile.metrics.annotation` package was influenced by the CDI extension for Dropwizard Metric project release 1.4.0.

Implementors can consult this project for implementation ideas.

See References for more information.

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
@ConcurrentGauge	M, C, T	Denotes a gauge which counts the parallel 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
@Metered	M, C, T	Denotes a meter, which tracks the frequency of invocations of the annotated object.	MetricUnits.PER_SECOND
@Metric	M, F, P	An annotation that contains the metadata information when requesting a metric to be injected or produced. This annotation can be used on fields of type `Meter`, `Timer`, `Counter`, and `Histogram`. For `Gauge`, the `@Metric` annotation can only be used on producer methods/fields.	MetricUnits.NONE
@Timed	M, C, T	Denotes a timer, which tracks duration of the annotated object.	MetricUnits.NANOSECONDS
@SimplyTimed	M, C, T	Denotes a simple timer, which tracks duration and invocations of the annotated object.	MetricUnits.NANOSECONDS

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

Annotation	Description	Default
@RegistryType	Qualifies the scope of Metric Registry to inject when injecting a MetricRegistry.	*application* (scope)

6.3.1. Fields

All annotations (Except `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 displayName`
: Optional. A human readable display name for metadata.

`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.

`boolean reusable`
: Denotes if a metric with a certain MetricID can be registered in more than one place. Does not apply to gauges.

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.

6.3.2. Annotated Naming Convention

Annotated metrics are registered into the *application* `MetricRegistry` with the name based on the annotation's `name` and `absolute` fields.

Example of annotated metric names

```
package com.example;

import javax.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
```

6.3.3. @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. The old behaviour pre 2.0 can now be achieved with `@ConcurrentGauge`.

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() {}

}
```

6.3.4. @ConcurrentGauge

An annotation for marking a method, constructor, or type as a parallel invocation counted. The semantics is such that upon entering a marked item, the parallel count is increased by one and

upon exit again decreased by one. The purpose of this annotation is to gauge the number of parallel invocations of the marked methods or constructors.

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 gauges, representing the current, previous minute maximum, and previous minute minimum values for the constructor using the Annotated Naming Convention.

Example of an annotated constructor

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

METHOD

When a non-private method is annotated, the implementation must register gauges, representing the current, previous minute maximum, and previous minute minimum values for the method using the Annotated Naming Convention.

Example of an annotated method

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

TYPE

When a type/class is annotated, the implementation must register gauges, representing the current, previous minute maximum, and previous minute minimum values for each of the constructors and non-private methods using the Annotated Naming Convention.

Example of an annotated type/class

```
@ConcurrentGauge
public class CounterBean {

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

}
```

6.3.5. @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;
}
```

6.3.6. @Metered

An annotation for marking a constructor or method as metered. The meter counts the invocations of the constructor or method and tracks how frequently they are called.

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 meter for the constructor using the Annotated Naming Convention. The meter is marked each time the constructor is invoked.

Example of an annotated constructor

```
@Metered
public MeteredBean() {
}
```

METHOD

When a non-private method is annotated, the implementation must register a meter for the method using the Annotated Naming Convention. The meter is marked each time the method is invoked.

Example of an annotated method

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

TYPE

When a type/class is annotated, the implementation must register a meter for each of the constructors and non-private methods using the Annotated Naming Convention. The meters are marked each time the corresponding constructor/method is invoked.

Example of an annotated type/class

```
@Metered
public class MeteredBean {

  public void meteredMethod1() {}
  public void meteredMethod2() {}

}
```

6.3.7. @SimplyTimed

An annotation for marking a constructor or method of an annotated object as simply timed. The metric of type SimpleTimer tracks the count of invocations of the annotated object and tracks how long it took the invocations to complete.

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 simple 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

```
@SimplyTimed
public SimplyTimedBean() {
}
```

METHOD

When a non-private method is annotated, the implementation must register a simple 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

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

TYPE

When a type/class is annotated, the implementation must register a simple 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 simple timer.

Example of an annotated type/class

```
@SimplyTimed
public class SimplyTimedBean {

  public void simplyTimedMethod1() {}
  public void simplyTimedMethod2() {}

}
```

6.3.8. @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() {}

}
```

6.3.9. @Metric

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

The implementation must support the following annotation targets:

- `FIELD`
- `METHOD`
- `PARAMETER`

The following lists the behavior for each annotation target.

FIELD

When a metric producer field is annotated, the implementation must register the metric to the application `MetricRegistry` (using the Annotated Naming Convention). If a metric with the given name already exist (created by another @Produces for example), an `java.lang.IllegalArgumentException` must be thrown.

Example of a producer field

```
@Produces
@Metric(name="hitPercentage")
@ApplicationScoped
Gauge<Double> hitPercentage = new Gauge<Double>() {

  @Override
  public Double getValue() {
      return hits / total;
  }
};
```

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 exist. If no metric exists with the given name then the implementation must produce and register the requested metric. `@Metric` can only be used on injected fields of type `Meter`, `Timer`, `Counter`, and `Histogram`.

Example of an injected field

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

METHOD

When a metric producer method is annotated, the implementation must register the metric produced by the method using the Annotated Naming Convention.

Example of a producer method

```
@Produces
@Metric(name = "hitPercentage")
@ApplicationScoped
protected Gauge<Double> createHitPercentage() {
    return new Gauge<Double>() {

      @Override
      public Double getValue() {
          return hits / total;
      }
    };
}
```

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();
}
```

6.4. 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. While using annotations is generally the preferred approach, 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.

6.4.1. 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
`counter(Metadata metadata, Tag… tags)`	Counter from given `Metadata` object with given tags
`concurrentGauge(String name)`	Concurrent gauge with given name and no tags
`concurrentGauge(String name, Tag… tags)`	Concurrent gauge with given name and tags
`concurrentGauge(Metadata metadata)`	Concurrent gauge from given `Metadata` object
`concurrentGauge(Metadata metadata, Tag… tags)`	Concurrent gauge 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
`meter(String name)`	Meter with given name and no tags
`meter(String name, Tag… tags)`	Meter with given name and tags
`meter(Metadata metadata)`	Meter from given `Metadata` object
`meter(Metadata metadata, Tag… tags)`	Meter 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
`simpleTimer(String name)`	SimpleTimer with given name and no tags
`simpleTimer(String name, Tag… tags)`	SimpleTimer with given name and tags
`simpleTimer(Metadata metadata)`	SimpleTimer from given `Metadata` object
`simpleTimer(Metadata metadata, Tag… tags)`	SimpleTimer from given `Metadata` object with given tags
`register(String name, T metric)`	Registers the given metric instance under the given name
`register(Metadata metadata, T metric)`	Registers the given metric instance using the given metadata object
`register(Metadata metadata, T metric, Tag… tags)`	Registers the given metric instance using the given metadata object and 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.

6.5. 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.

6.5.1. 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

6.6. 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 scope (*application*, *base*, and *vendor*). When metrics are registered using annotations, the metrics are registered in the *application* `MetricRegistry` (and thus the *application* scope).

When injected, the `@RegistryType` is used as a qualifier to selectively inject either the `APPLICATION`, `BASE`, or `VENDOR` registry. If no qualifier is used, the default `MetricRegistry` returned is the `APPLICATION` registry.

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.

6.6.1. @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.

6.6.2. Application Metric Registry

The implementation must produce the *application* `MetricRegistry` when no `RegistryType` is provided (`@Default`) or when the `RegistryType` is `APPLICATION`.

Example of the application injecting the application registry

```
@Inject
MetricRegistry metricRegistry;
```

is equivalent to

```
@Inject
@RegistryType(type=MetricRegistry.Type.APPLICATION)
MetricRegistry metricRegistry;
```

6.6.3. Base Metric Registry

The implementation must produce the *base* `MetricRegistry` when the `RegistryType` is `BASE`. The *base* `MetricRegistry` must contain the required metrics specified in Required Metrics.

Example of the application injecting the base registry

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

6.6.4. Vendor Metric Registry

The implementation must produce the *vendor* `MetricRegistry` when the `RegistryType` is `VENDOR`. The *vendor* `MetricRegistry` must contain any vendor specific metrics.

Example of the application injecting the vendor registry

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

6.6.5. 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")
    .withType(MetricType.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.

Chapter 7. Appendix

7.1. Alternatives considered

Jolokia JMX-HTTP bridge. Using this for application specific metrics would require those metrics to be exposed to JMX first, which many users are not familiar with.

7.2. References

Dropwizard Metrics 3.2.3

CDI extension for Dropwizard Metrics 1.4.0

HTTP return codes

UoM, JSR 363

Metrics 2.0

7.3. 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"
    displayName: "Current Thread count"
  - 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.

7.4. Example Metric Registry Factory

Sample skeleton factory class to produce MetricRegistry via CDI

```
@ApplicationScoped
public class MetricRegistryFactory {

    @Produces
    public static MetricRegistry getDefaultRegistry() {
        return getApplicationRegistry();
    }

    @Produces
    @RegistryType(type = Type.APPLICATION)
    public static MetricRegistry getApplicationRegistry() {
        // Returns the static instance of the Application MetricRegistry
        [...]
    }

    @Produces
    @RegistryType(type = Type.BASE)
    public static MetricRegistry getBaseRegistry() {
        // Returns the static instance of the Base MetricRegistry
        [...]
    }

    @Produces
    @RegistryType(type = Type.VENDOR)
    public static MetricRegistry getVendorRegistry() {
        // Returns the static instance of the Vendor MetricRegistry
        [...]
    }

}
```

7.5. Migration hints

7.5.1. To version 2.0

@Counted

The `@Counted` annotation has changed. Users of the previous `@Counted` annotation were surprised to learn that by default counters were not monotonic. Also, the OpenMetrics format expects all counters to be monotonic. To migrate:

- Replace `@Counted()` or `@Counted(monotonic=false)` with `@ConcurrentGauge`. A set of gauges will be created in the OpenMetrics output for each `@ConcurrentGauge`.
- Replace `@Counted(monotonic=true)` with `@Counted` (monotonic flag is gone)

This change has also had an impact on the `Counter` interface to basically follow the above change:

- Modify code which uses `Counter.dec()` to use a `Gauge` or `ConcurrentGauge`.

Some base metrics' types have changed from `Counter` to `Gauge` since counters must now count monotonically. Update code or dashboards that use the following metrics:

- thread.count
- thread.daemon.count
- classloader.currentLoadedClass.count
- thread.max.count

Some base metrics' names have changed to follow the convention of ending the name of accumulating counters with `total`. Update code or dashboards that use the following metrics:

- gc.count → gc.total
- classloader.currentLoadedClass.count → classloader.loadedClasses.count (changed to stay consistent with other classloader metric names)
- classloader.totalLoadedClass.count → classloader.loadedClasses.total
- classloader.totalUnloadedClass.count → classloader.unloadedClasses.total

Chapter 8. Major changes to previous versions

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

- Changes in 2.3
  - Introduced the simple timer (`@SimplyTimed`) metric.
  - 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
  - Added ProcessCpuTime as a new optional base metric.
  - Added `withOptional*` methods to the `MetadataBuilder`, they don't fail when null values are passed to them
  - Added the `MetricID.getTagsAsArray()` method to the API.
  - Added the method `MetricType.fromClassName`
  - Improved TCK - Use WebArchive for deployment
- Changes in 2.2
  - Reverted a problematic change from 2.1 where Gauges were required to return subclasses of `java.lang.Number`
- Changes in 2.1
  - (2.1.1) Added ProcessCpuTime as a new optional base metric.
  - Clarified that metric registry implementations are required to be thread-safe.
  - 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]
  - Clarified that implementations can, for JSON export of scopes containing no metrics, omit them, or that they can be present with an empty value.
  - Clarified that metrics should not be created for private methods when a class is annotated (the TCK asserted this in 2.0 anyway)
  - TCKs are updated to use RestAssured 4.0
  - Added the `reusable(boolean)` method for MetadataBuilder
  - Explicitly excluded the transitive dependency on `javax.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.
  - Added some text to the specification about programmatic creation of metrics (without annotations)
- Changes in 2.0
  - (2.0.3) Added ProcessCpuTime as a new optional base metric.
  - ϟ Refactoring of Counters, as the old `@Counted` was misleading in practice.
    - 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.
- See Migration hints about migration of applications using MicroProfile Metrics.

- Removed unnecessary `@InterceptorBinding` annotation from `org.eclipse.microprofile.metrics.annotation.Metric`.
- ϟ Removed deprecated `org.eclipse.microprofile.metrics.MetricRegistry.register(String name, Metric, Metadata)`
- ϟ `Metadata` is now immutable and built via a `MetadataBuilder`.
- Introduced a Tag object which represents a singular tag key/value pair.
- ϟ Metrics are now uniquely identified by a MetricID (combination of the metric's name and tags).
- MetricFilter modified to filter with MetricID instead of name
- The 'Metadata' is mapped to a unique metric name in the `MetricRegistry` and this relationship is immutable.
- Tag key names for labels are restricted to match the regex `[a-zA-Z_][a-zA-Z0-9_]*`.
- Tag values defined through MP_METRICS_TAGS must escape equal signs `=` and commas `,` with a backslash `\`.
- 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.
- OpenMetrics format - formerly called Prometheus format
  - Reserved characters in OpenMetrics format must be escaped.
  - ϟ In OpenMetrics output format, the separator between scope and metric name is now a `_` instead of a `:`.
  - ϟ Metric names with camelCase are no longer converted to snake_case for OpenMetrics output.
- ϟ The default value of the `reusable` attribute for metric objects created programmatically (not via annotations) is now `true`
- ϟ Some base metrics' names have changed to follow the convention of ending the name of accumulating counters with `total`.
- ϟ Some base metrics' types have changed from Counter to Gauge since Counters must now count monotonically.
- 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.
- Added a set of recommendations how application servers with multiple deployed

applications should behave if they support MP Metrics.

- Changes in 1.1
  - Improved TCK
  - `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).
  - 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.