Keeping Count In Jakarta EE Applications With MicroProfile Metrics

The Payara® Platform - Production-Ready, Cloud Native and Aggressively Compatible.

User Guide

Contents

Guide Updated: **May 2023**

Deployed, user-facing applications need to be monitored to ensure they are delivering as expected. Cloud-native applications deployed to complex cloud hosting environments need to be monitored even more because any unexpected deviation from preset expectations could mean significant extra hosting bills.

The way to monitor deployed, running applications is through the use of application metrics. Different metrics such as how many times a method has been called, how many seconds did a given method take to return are all different kinds of metrics that can give insights into how an application is performing.

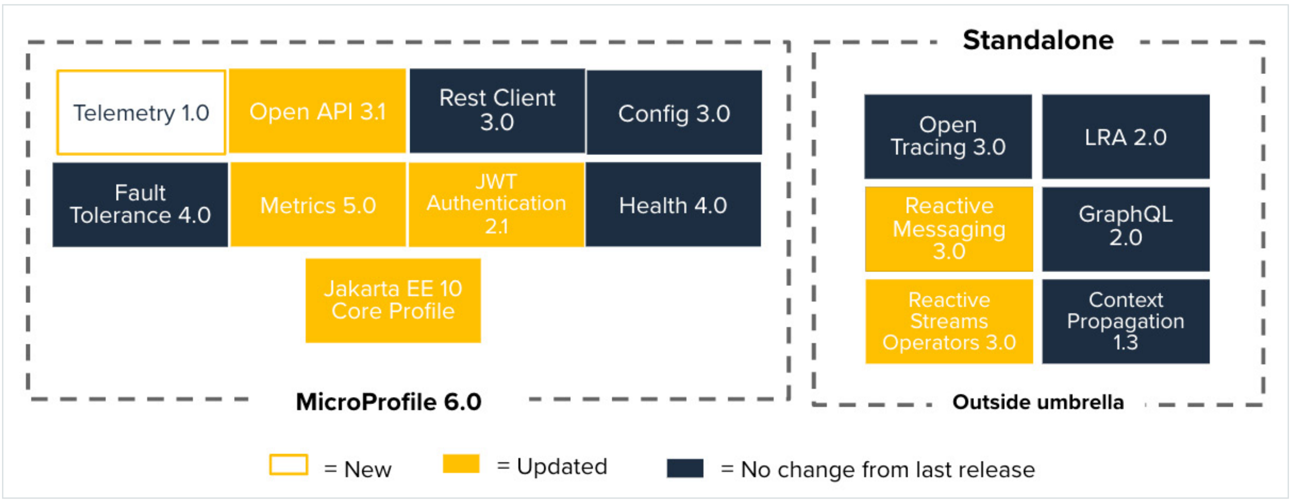
The MicroProfile Metrics API provides a set of API constructs for exposing different types and kinds of application metrics. With this API, you can expose application, implementation runtime and JVM metrics through annotations and a Java API in the OpenMetrics format, consumable by monitoring applications like Prometheus.

This guide covers the Metrics specification within a Jakarta EE context. By the end of the chapter, you will be able to use the Metrics API to expose different kinds of metrics in your cloud-native Jakarta EE application.

What is MicroProfile?

MicroProfile is a community driven initiative, built on top of the Jakarta EE Core Profile, that is a collection of abstract specs that form a complete solution to developing cloud native, Jakarta EE microservices. The goal is to create a set of APIs that abstracts you from their implementations so that you can create highly portable microservices across vendors.

The current release is version 6.0 which is a major release that includes MicroProfile Config 3.0, MicroProfile Fault Tolerance 4.0, MicroProfile Health 4.0, MicroProfile Metrics 5.0, and MicroProfile Rest Client 3.0. MicroProfile 6.0 is built on top of the Core Profile of Jakarta EE, a slimmed down version of Jakarta EE "that contains a set of Jakarta EE Specifications targeting smaller runtimes suitable for microservices and ahead-of-time compilation." The Core Profile of Jakarta EE was released as part of Jakarta EE 10. MicroProfile 6.0 is incompatible with versions of Jakarta EE below 10.



As abstract specifications, the various implementations are free to implement the base specs and add custom features on top. Payara Server is one of the popular implementations of the MicroProfile spec and adds quite a number of custom features on top of the base specs. You can download a free trial of Payara Enterprise here to follow along with the rest of the guide.

Getting Started with MicroProfile

To get started with the MicroProfile API, you need to include it as a dependency in your project as shown below.

```
<dependency>
    <groupId>org.eclipse.microprofile</groupId>
    <artifactId>microprofile</artifactId>
    <version>6.0</version>
    <type>pom</type>
    <scope>provided</scope>
</dependency>
```

With the MicroProfile API dependency in place, you have access to all the APIs of the project. In our case, the Payara Server will provide the implementation for us.

Major Changes In Metrics 5.0

As shown in the image above, the current version of the Metrics API released as part of MicroProfile 6 is Metrics 5.0, which features some notable breaking changes. These are

- Removal of SimpleTimer class and SimplyTimed annotation
- Removal of ConcurrentGauge class and ConcurrentGauge annotation
- Removal of Meter class and Metered annotation
- Removal of Metered interface
- Removal of MetricType enum
- Removal of JSON data format for returned metrics
- Removed prepending metrics with the scope in favour of using mp_scope tag

You should keep these changes in mind when upgrading from previous versions of this specification.

What is the Metrics API?

To understand what the Metrics API is, let us seek to understand the issues it helps address. All deployed applications need to be monitored to ensure they are performing to expectations. This is especially critical in a cloud native microservices environment. Cloud-native applications deployed to complex cloud hosting environments need to be monitored even more because any unexpected deviation from preset expectations could mean significant extra hosting bills.

Typical metrics you would want to gather about your application could include how long a given method takes to execute, how many times a given method is invoked, how many times a given class is instantiated, and what the average concurrent access is to a given method per period, among other examples.

These are all metrics that need to be collected and monitored to ensure they meet set standards of an organisation. Any deviation beyond a certain threshold will then need to be investigated and remedied. The MicroProfile Metrics API seeks to help you gather actionable metrics not just about your application, but also your application server and even the JVM.

The Metrics API gives you a set of annotations and Java APIs that you can use to easily expose application metrics in the OpenMetrics format that can be consumed by systems monitoring and alerting toolkits like Prometheus.

Why A Separate Metrics API?

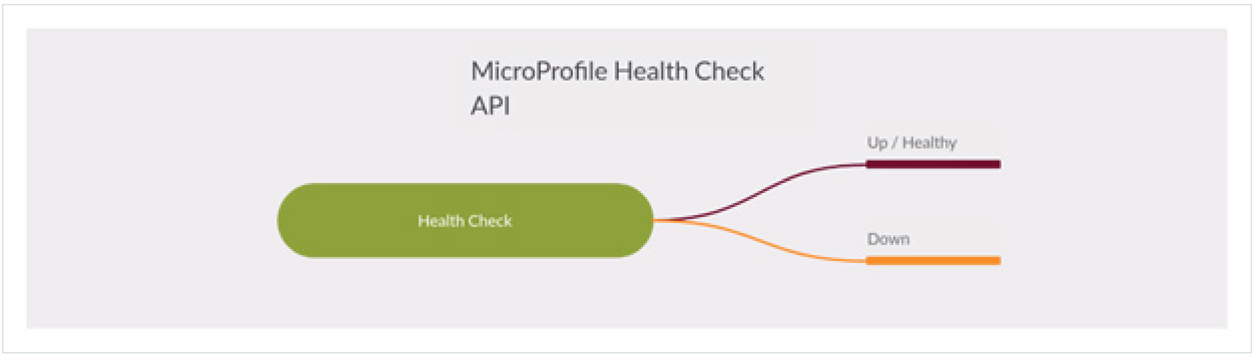
You may be wondering why, in the light of the popularity of libraries like Micrometer and OpenTelemetry, the MicroProfile project maintains a separate metrics API. According to the specification, the Metrics API:

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

Consequently, the Metrics API had to be slimmed down to provide only core metrics functionality. This resulted in the many breaking changes in recent releases.

Metrics vs Health Check

You might be wondering what the point of a separate Metrics API is when the MicroProfile project already comes with a Health Check API. Both APIs have to do with monitoring the health of your deployed applications. However, that is as far as the similarities go. The Health Check API is used to check the health of an application in a binary way - "is my application up or not?" The image below depicts the core function of the Health Check API.



The Health Check API simply indicates if an application is up or not, and whether it is ready to accept connections.

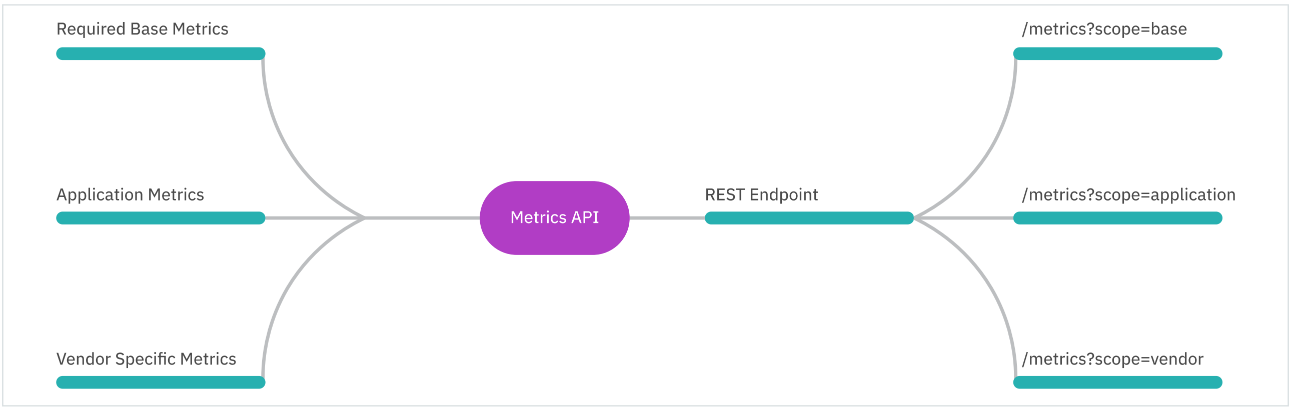
The MicroProfile Metrics API, on the other hand, helps you monitor the extent to which an application is healthy, or not. For instance, you may have a method that takes 15 seconds to respond to a request. However, your expectation is that this particular method should take no more than 9 seconds to do so. In this case, even though your method does respond to requests, it is not necessarily healthy. Further digging might be needed to remedy this situation.

The Metrics API gives you various annotations and a Java API to be able to collect fine-grained metrics about your deployed microservices. As shown in the above image, it comes with the following types of metrics:

- **Counter:** a monotonically increasing numeric value (e.g. total number of requests received).
- **Gauge:** a metric that is sampled to obtain its value (e.g. CPU temperature or disk usage).
- **Timer:** a metric which aggregates timing durations and provides duration statistics, plus throughput statistics.
- **Histogram:** a metric which calculates the distribution of a value.

The Structure of the Metrics API

The MicroProfile Metrics API can be viewed as comprising two broad components as demonstrated in the image below.



The left side of the image shows the classes of metrics that can be collected. These categories of metrics are referred to as scopes. So there are 3 metric scopes - required base metrics, application metrics and vendor specific metrics.

Required Base Metrics

Base metrics refers to a set of metrics that MicroProfile compliant servers may provide. These metrics are generally related to the underlying JVM and operating system. Examples of these metrics are used heap memory, committed heap memory, maximum heap memory, and available processors.

Application Metrics

Application metrics refers to the set of metrics that you as the application developer would like to collect about your application. Because this scope is absolutely application dependent, you have a Java API to use to craft whatever metrics you would want to gather. Examples of application specific metrics you can gather are how many times a given method is invoked, how long does a method take to respond to a request among other others.

Vendor Specific Metrics

Vendor specific metrics are custom metrics that a MicroProfile compliant runtime provides on top of the required base metrics. For example, the Payara Server allows you to expose JMX Mbeans as custom vendor metrics by providing a custom metrics.xml file.

Exposing Metrics

The right side of the above image shows a rest endpoint that exposes the metrics collected. By default, MicroProfile Metrics are available on the /metrics resource path of the root server. So for instance, an application hosted on the domain foobazz.com that has MicroProfile Metrics enabled will have https://foobazz.com/metrics. In your local development environment using the Payara Server, you can access Metrics via htttp://localhost:8080/metrics.

Metrics Data Format

By default, a GET HTTP request with content type set to text/plain to the /metrics resource returns metrics data in the Prometheus text format. The metrics endpoint can also return the metrics data in the OpenMetrics format if the HTTP Accept header best matches application/openmetrics-text; version=1.0.0.

Metrics Scopes Paths

The 3 metrics scopes - base, application. and custom vendor metrics - all have their respective rest endpoints for accessing metrics specific to them. They can all be accessed from the base /metric path.

- /metrics?scope=base for required base metrics
- /metrics?scope=application for application specific metrics
- /metrics?scope=vendor for custom vendor metrics

Metrics in Practice

Before looking at the various annotations available to you from the Metrics API, let us first take a look some parameters that are common to all Metrics annotations:

- 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.
- String[] tags - An array of custom application metric tags

These parameters determine the metadata for a given annotation.

The MetricsRegistry

The MetricsRegistry is a registry of all metrics annotations and metadata you create in your application. There is a MetricsRegistry for each of the metrics scopes of base, application and vendor. Metrics are identified in the MetricsRegistry by their metrics ID. A metric ID is a combination of the name and tags (optional) of a given metric. For now, just think of the MetricsRegistry as the black box that contains all the metrics you declare in your application.

Let us now see how you can gather application metrics using the various annotations and Java API that the metrics API provides.

@Counted

The @Counted annotation is the simplest metrics annotation. It simply counts how many times an annotated artefact is invoked/instantiated. For instance the method in the code snippet below is annotated @Counted with no parameters.

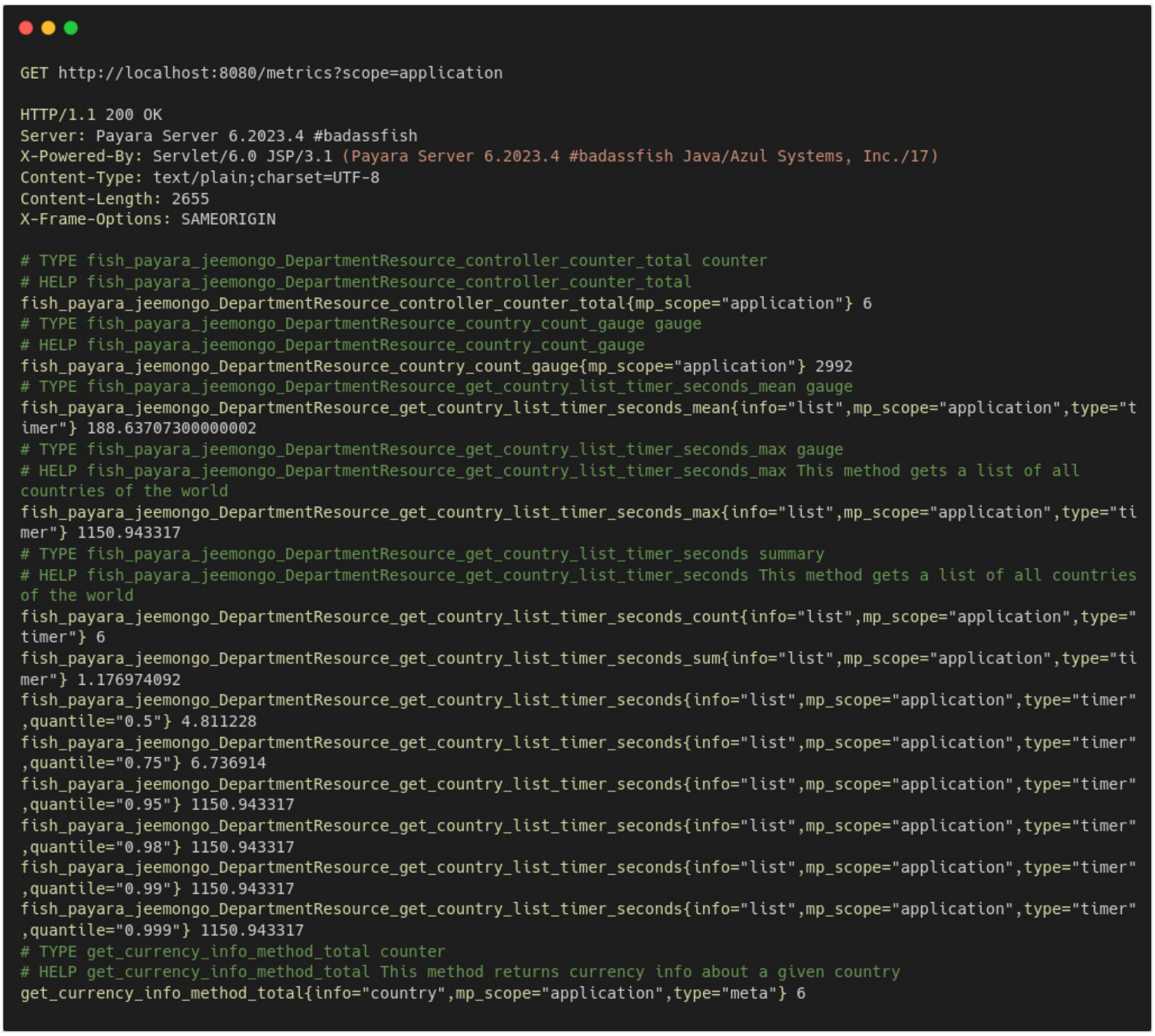
```
@GET
@Path(“currency/{country}”)
@Counted
public CurrencyInfo getCurrencyInfo(@PathParam(“country”) String country) {
    return controller.getCurrencyInfo(country);
}
```

The getCurrencyInfo method is annotated with @Counted, meaning for every invocation of this method, the counter will be incremented by 1. The name of this metric will be the fully qualified method name. Now let us add some custom metadata to the @Counted annotation.

```
@GET
@Path("currency/{country}")
@Counted(name = "get_currency_info_method",
        absolute = true,
        description = "This method returns currency info about a given
country",
        tags = { "info=country", "type=meta" })
public CurrencyInfo getCurrencyInfo(@PathParam("country") String country) {
    return controller.getCurrencyInfo(country);
}
```

In the above code snippet, we have added some metadata to the @Counted annotation. The name field is now set to get_currency_info_method. This name is set to absolute. This means the name of this particular metric will be just what is passed to the name field instead of the package name + class name + method name. The description has been set to a brief text about the function of this metric. We also pass in some tags to the tags field of @Counted. This is going to be used as part of the ID of this particular metric.

The above metric can be accessed from the /metrics?scope=application endpoint when this code is deployed. By default, a GET request to this endpoint returns data in the Prometheus text format as shown below.



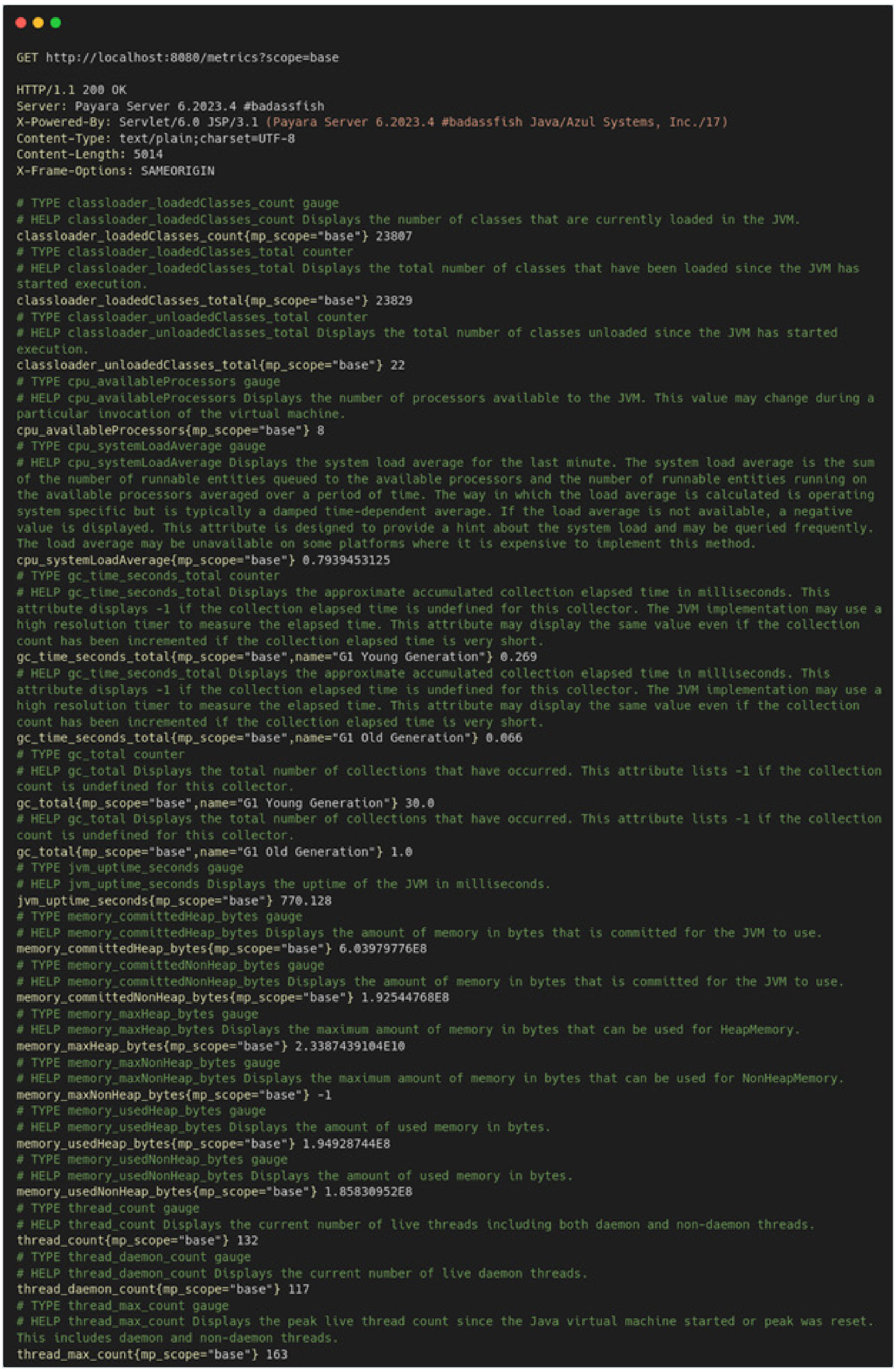
```
GET http://localhost:8080/metrics?scope=application

HTTP/1.1 200 OK
Server: Payara Server 6.2023.4 #badassfish
X-Powered-By: Servlet/6.0 JSP/3.1 (Payara Server 6.2023.4 #badassfish Java/Azul Systems, Inc./17)
Content-Type: text/plain;charset=UTF-8
Content-Length: 2655
X-Frame-Options: SAMEORIGIN

# TYPE fish_payara_jeemongo_DepartmentResource_controller_counter_total counter
# HELP fish_payara_jeemongo_DepartmentResource_controller_counter_total
fish_payara_jeemongo_DepartmentResource_controller_counter_total{mp_scope="application"} 6
# TYPE fish_payara_jeemongo_DepartmentResource_country_count_gauge gauge
# HELP fish_payara_jeemongo_DepartmentResource_country_count_gauge
fish_payara_jeemongo_DepartmentResource_country_count_gauge{mp_scope="application"} 2992
# TYPE fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds_mean gauge
fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds_mean{info="list",mp_scope="application",type="t
imer"} 188.63707300000002
# TYPE fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds_max gauge
# HELP fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds_max This method gets a list of all
countries of the world
fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds_max{info="list",mp_scope="application",type="ti
mer"} 1150.943317
# TYPE fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds summary
# HELP fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds This method gets a list of all countries
of the world
fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds_count{info="list",mp_scope="application",type="
timer"} 6
fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds_sum{info="list",mp_scope="application",type="ti
mer"} 1.176974092
fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds{info="list",mp_scope="application",type="timer"
,quantile="0.5"} 4.811228
fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds{info="list",mp_scope="application",type="timer"
,quantile="0.75"} 6.736914
fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds{info="list",mp_scope="application",type="timer"
,quantile="0.95"} 1150.943317
fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds{info="list",mp_scope="application",type="timer"
,quantile="0.98"} 1150.943317
fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds{info="list",mp_scope="application",type="timer"
,quantile="0.99"} 1150.943317
fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds{info="list",mp_scope="application",type="timer"
,quantile="0.999"} 1150.943317
# TYPE get_currency_info_method_total counter
# HELP get_currency_info_method_total This method returns currency info about a given country
get_currency_info_method_total{info="country",mp_scope="application",type="meta"} 6
```

As you can see from the above image, there is some information being emitted from our deployed sample code. The one of interest to us now is the second uncommented line application_get_currency_info_method_total{“info=country”, mp_scope=”application”, “type=meta”} 6. This line refers to our @Counted annotated method. The number 6 means this method has been invoked 6 times in total.

If you examine all the metrics in the above image, you can see they are all tagged {mp_scope=”application”}. This is the way to identify that these metrics are from the application scope. Similarly, a request to /metrics?scope=base returns metrics data tagged {mp_scope=”base”}.

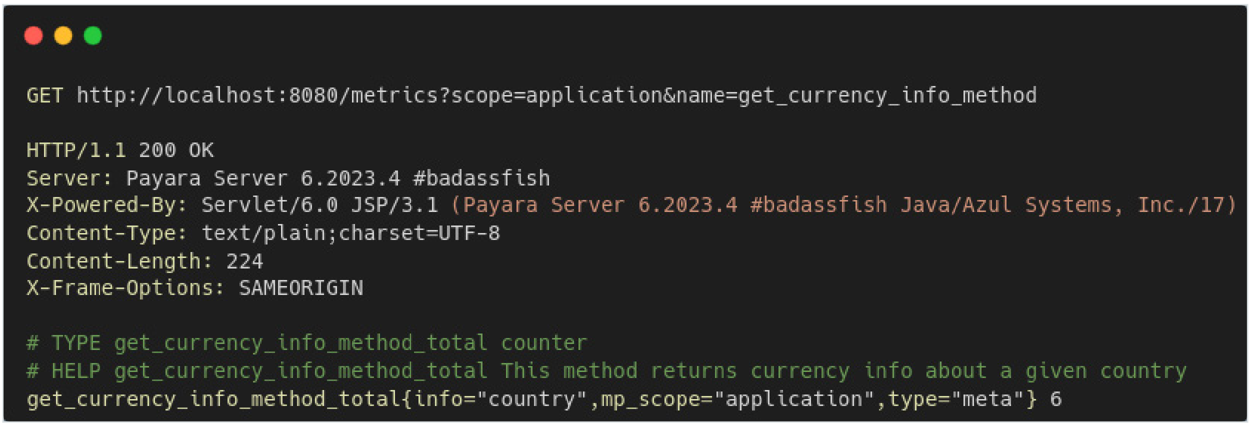


```
GET http://localhost:8080/metrics?scope=base

HTTP/1.1 200 OK
Server: Payara Server 6.2023.4 #badassfish
X-Powered-By: Servlet/6.0 JSP/3.1 (Payara Server 6.2023.4 #badassfish Java/Azul Systems, Inc./17)
Content-Type: text/plain;charset=UTF-8
Content-Length: 5014
X-Frame-Options: SAMEORIGIN

# TYPE classloader_loadedClasses_count gauge
# HELP classloader_loadedClasses_count Displays the number of classes that are currently loaded in the JVM.
classloader_loadedClasses_count{mp_scope="base"} 23807
# TYPE classloader_loadedClasses_total counter
# HELP classloader_loadedClasses_total Displays the total number of classes that have been loaded since the JVM has
started execution.
classloader_loadedClasses_total{mp_scope="base"} 23829
# TYPE classloader_unloadedClasses_total counter
# HELP classloader_unloadedClasses_total Displays the total number of classes unloaded since the JVM has started
execution.
classloader_unloadedClasses_total{mp_scope="base"} 22
# TYPE cpu_availableProcessors gauge
# HELP cpu_availableProcessors Displays the number of processors available to the JVM. This value may change during a
particular invocation of the virtual machine.
cpu_availableProcessors{mp_scope="base"} 8
# TYPE cpu_systemLoadAverage gauge
# HELP cpu_systemLoadAverage 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.
cpu_systemLoadAverage{mp_scope="base"} 0.7939453125
# TYPE gc_time_seconds_total counter
# HELP gc_time_seconds_total Displays the approximate accumulated collection elapsed time in milliseconds. This
attribute displays -1 if the collection elapsed time is undefined for this collector. The JVM 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.
gc_time_seconds_total{mp_scope="base",name="G1 Young Generation"} 0.269
# HELP gc_time_seconds_total Displays the approximate accumulated collection elapsed time in milliseconds. This
attribute displays -1 if the collection elapsed time is undefined for this collector. The JVM 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.
gc_time_seconds_total{mp_scope="base",name="G1 Old Generation"} 0.066
# TYPE gc_total counter
# HELP gc_total Displays the total number of collections that have occurred. This attribute lists -1 if the collection
count is undefined for this collector.
gc_total{mp_scope="base",name="G1 Young Generation"} 30.0
# HELP gc_total Displays the total number of collections that have occurred. This attribute lists -1 if the collection
count is undefined for this collector.
gc_total{mp_scope="base",name="G1 Old Generation"} 1.0
# TYPE jvm_uptime_seconds gauge
# HELP jvm_uptime_seconds Displays the uptime of the JVM in milliseconds.
jvm_uptime_seconds{mp_scope="base"} 770.128
# TYPE memory_committedHeap_bytes gauge
# HELP memory_committedHeap_bytes Displays the amount of memory in bytes that is committed for the JVM to use.
memory_committedHeap_bytes{mp_scope="base"} 6.03979776E8
# TYPE memory_committedNonHeap_bytes gauge
# HELP memory_committedNonHeap_bytes Displays the amount of memory in bytes that is committed for the JVM to use.
memory_committedNonHeap_bytes{mp_scope="base"} 1.92544768E8
# TYPE memory_maxHeap_bytes gauge
# HELP memory_maxHeap_bytes Displays the maximum amount of memory in bytes that can be used for HeapMemory.
memory_maxHeap_bytes{mp_scope="base"} 2.3387439104E10
# TYPE memory_maxNonHeap_bytes gauge
# HELP memory_maxNonHeap_bytes Displays the maximum amount of memory in bytes that can be used for NonHeapMemory.
memory_maxNonHeap_bytes{mp_scope="base"} -1
# TYPE memory_usedHeap_bytes gauge
# HELP memory_usedHeap_bytes Displays the amount of used memory in bytes.
memory_usedHeap_bytes{mp_scope="base"} 1.94928744E8
# TYPE memory_usedNonHeap_bytes gauge
# HELP memory_usedNonHeap_bytes Displays the amount of used memory in bytes.
memory_usedNonHeap_bytes{mp_scope="base"} 1.85830952E8
# TYPE thread_count gauge
# HELP thread_count Displays the current number of live threads including both daemon and non-daemon threads.
thread_count{mp_scope="base"} 132
# TYPE thread_daemon_count gauge
# HELP thread_daemon_count Displays the current number of live daemon threads.
thread_daemon_count{mp_scope="base"} 117
# TYPE thread_max_count gauge
# HELP thread_max_count Displays the peak live thread count since the Java virtual machine started or peak was reset.
This includes daemon and non-daemon threads.
thread_max_count{mp_scope="base"} 163
```

Using the default REST endpoint, you could also get back only the metric for the annotated method using the name of the metric by accessing /metrics?scope=application&name=get_currency_info_method as shown below.



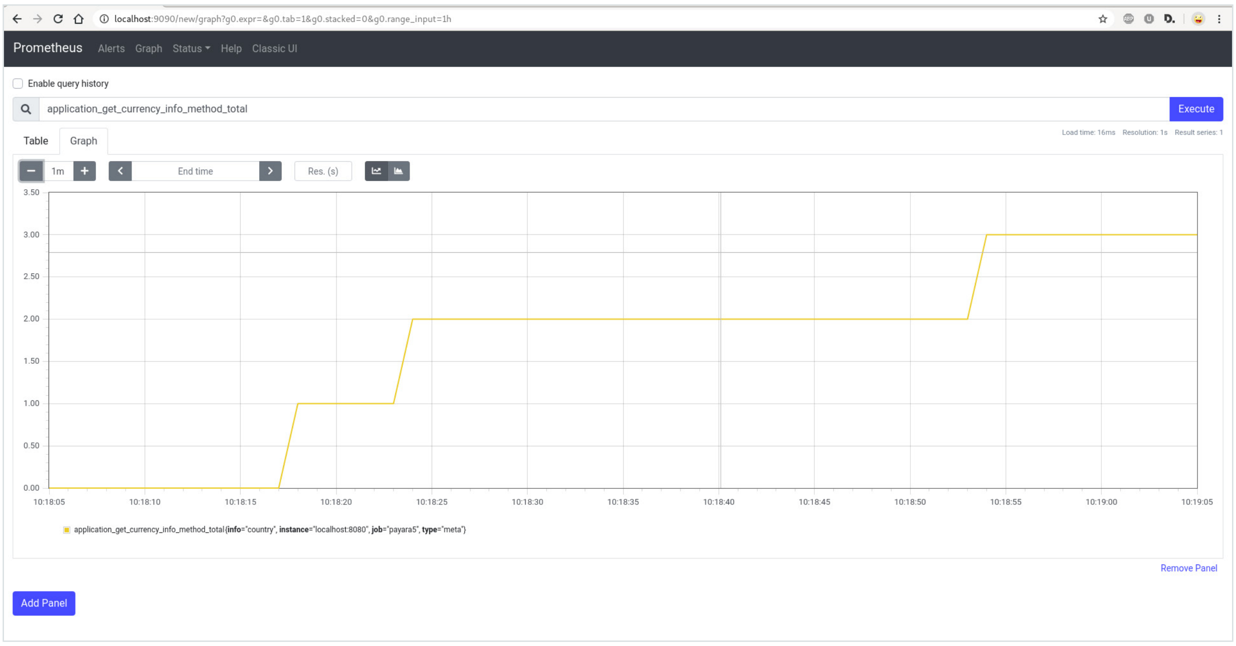
```
GET http://localhost:8080/metrics?scope=application&name=get_currency_info_method

HTTP/1.1 200 OK
Server: Payara Server 6.2023.4 #badassfish
X-Powered-By: Servlet/6.0 JSP/3.1 (Payara Server 6.2023.4 #badassfish Java/Azul Systems, Inc./17)
Content-Type: text/plain;charset=UTF-8
Content-Length: 224
X-Frame-Options: SAMEORIGIN

# TYPE get_currency_info_method_total counter
# HELP get_currency_info_method_total This method returns currency info about a given country
get_currency_info_method_total{info="country",mp_scope="application",type="meta"} 6
```

The last line in the image above shows the same 6 as being the number of times our getCurrencyInfo method has been invoked.

The metrics exposed by the Metrics API can be used as is with monitoring applications like Prometheus. For instance, the image below shows a graph of invocations of the getCurrencyInfo as mapped by Prometheus.



localhost:9090/new/graph?g0.expr=&g0.tab=1&g0.stacked=0&g0.range_input=1h

application_get_currency_info_method_total

Load time: 16ms Resolution: 1s Result series: 1

application_get_currency_info_method_total{info="country", instance="localhost:8080", job="payara5", type="meta"}

The @Counted metric annotation can be used on a method, class or constructor. When used on a class, a separate metric is created for both class instantiation and method invocations. When used on a constructor, the metric counts the number of times that constructor is invoked. In our example above, we used it on a method. So only invocations of that method will be counted.

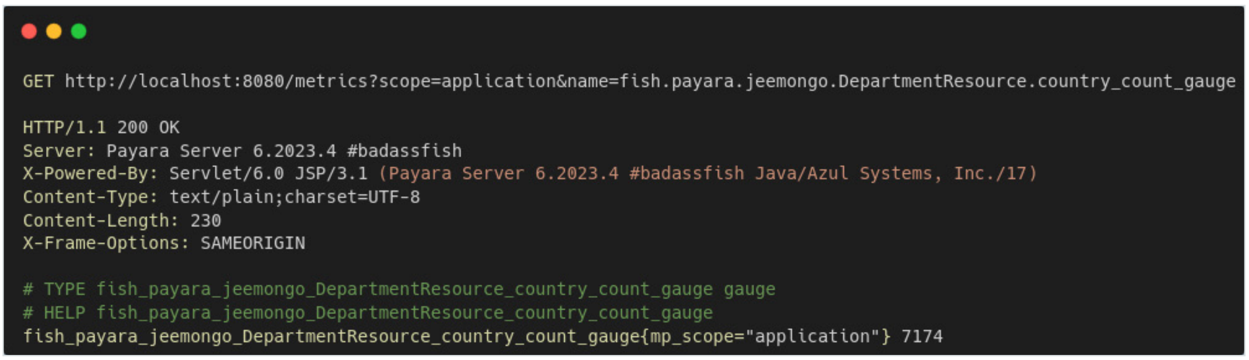
@Gauge

This metric is used to sample the value of an object. A typical example would be in a database application, you may want to find how many sign ups you have to date. You would typically annotate a method with @Gauge in order to sample the returned value at any given time. The @Gauge metric can be used only on a method, with the return value of the method acting as the value of the metric for each invocation of the metric.

In the code snippet below, the getCountOfCountries method is annotated with @Gauge in order to sample the number of countries returned from our service.

```
@GET
@Path("countries/count")
@Produces(MediaType.TEXT_PLAIN)
@Gauge(name = "country_count_gauge", unit = MetricUnits.NONE)
public Long getCountOfCountries() {
    return controller.getCountriesCount();
}
```

The method is annotated with @Gauge, setting the name to country_count_gauge. Setting the metric unit parameter for @Gauge is mandatory, so we set it to MetricUnits.NONE which defaults to integer count. This unit lends itself better to the kind of metric we are returning as compared to all the available ones from the MetricsUnits class. As the metric is not set to absolute, it can be directly accessed through the fully qualified class name. A sample request returns the total count as shown below.



```
GET http://localhost:8080/metrics?scope=application&name=fish.payara.jeemongo.DepartmentResource.country_count_gauge

HTTP/1.1 200 OK
Server: Payara Server 6.2023.4 #badassfish
X-Powered-By: Servlet/6.0 JSP/3.1 (Payara Server 6.2023.4 #badassfish Java/Azul Systems, Inc./17)
Content-Type: text/plain;charset=UTF-8
Content-Length: 230
X-Frame-Options: SAMEORIGIN

# TYPE fish_payara_jeemongo_DepartmentResource_country_count_gauge gauge
# HELP fish_payara_jeemongo_DepartmentResource_country_count_gauge
fish_payara_jeemongo_DepartmentResource_country_count_gauge{mp_scope="application"} 7174
```

@Timed

The @Timed annotation is used to track how frequently an annotated object is invoked, and tracks how long it took the invocations to complete. It can be used on a method, class or constructor. In the sample code, we would like to time the execution of the getCountryList method. This method could potentially take some time to execute given the computationally expensive nature of the work it does. We would like to get detailed execution statistics using the @Timed annotation as shown below.

```
@GET
@Path("countries")
@Timed(name = "get_country_list_timer",
        description = "This method gets a list of all countries of the
world",
        unit = MetricUnits.MICROSECONDS,
        tags = { "info=list", "type=timer" })

public CompletionStage<List<String>> getCountryList() {
    return controller.getCountryList();
}
```

The getCountryList is annotated with @Timed annotation. The name of the metric is set to get_country_list_timer. The description, metric unit and tags are also set. This metric can be accessed using the fully qualified class name and the given name for the metric - get_country_list_timer as shown below.

```
GET http://localhost:8080/metrics?
scope=application&name=fish.payara.jeemongo.DepartmentResource.get_country_list_timer

HTTP/1.1 200 OK
Server: Payara Server 6.2023.4 #badassfish
X-Powered-By: Servlet/6.0 JSP/3.1 (Payara Server 6.2023.4 #badassfish Java/Azul Systems, Inc./17)
Content-Type: text/plain;charset=UTF-8
Content-Length: 1954
X-Frame-Options: SAMEORIGIN

# TYPE fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds_mean gauge
fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds_mean{info="list",mp_scope="application",type=
"timer"} 129.470822
# TYPE fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds_max gauge
# HELP fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds_max This method gets a list of all
countries of the world
fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds_max{info="list",mp_scope="application",type="
timer"} 809.254433
# TYPE fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds summary
# HELP fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds This method gets a list of all
countries of the world
fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds_count{info="list",mp_scope="application",type
="timer"} 8
fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds_sum{info="list",mp_scope="application",type="
timer"} 1.060660462
fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds{info="list",mp_scope="application",type="time
r",quantile="0.5"} 4.903941
fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds{info="list",mp_scope="application",type="time
r",quantile="0.75"} 5.1838310000000005
fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds{info="list",mp_scope="application",type="time
r",quantile="0.95"} 809.254433
fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds{info="list",mp_scope="application",type="time
r",quantile="0.98"} 809.254433
fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds{info="list",mp_scope="application",type="time
r",quantile="0.99"} 809.254433
fish_payara_jeemongo_DepartmentResource_get_country_list_timer_seconds{info="list",mp_scope="application",type="time
r",quantile="0.999"} 809.254433
```

As you can see, the @Timed annotation generates detailed statistical insights into the execution time of the method.

Injecting Metrics

The metrics you have seen so far have all been used in the form of annotations. The Metrics API also supports injecting metrics types into fields, methods or parameters and then manually use those metrics using the @Metric annotation. For instance, in our sample code, we would want to inject a counter into our controller class Controller.java and manually increment method invocations. The whole workflow is shown below.

```java
@Inject
@Metric(name = “controller_counter”)
private Counter count;

public CurrencyInfo getCurrencyInfo(String country) {
    count.inc();
    return geoFetch.getCurrencyInfo(country);
}
```

The Controller.java class annotates field count of type Counter with @Inject and @Metric, respectively. This causes the Metrics implementation to inject a contextual instance of the Counter type into the count field. This is done by getting an already existing instance from the MetricsRegistry. if one exists, or creating and registering one if none exists in the MetricsRegistry. The getCurrencyInfo method calls the inc() on the count object, which increments the counter by 1 for every invocation of the method.

When we used the @Counted annotation, we were able to pass some metadata to be associated with the counter. The @Metric annotation used in the above code also does support the same metadata as the @Counted annotation. Because we passed in a name for the count counter without specifying it as absolute, we can access this metric from the REST endpoint /metrics?scope=application&name=fish.payara.control.ft.DepartmentResource.controller_counter as shown below.

```
GET http://localhost:8080/metrics?scope=application&name=fish.payara.jeemongo.DepartmentResource.controller_counter

HTTP/1.1 200 OK
Server: Payara Server 6.2023.4 #badassfish
X-Powered-By: Servlet/6.0 JSP/3.1 (Payara Server 6.2023.4 #badassfish Java/Azul Systems, Inc./17)
Content-Type: text/plain;charset=UTF-8
Content-Length: 244
X-Frame-Options: SAMEORIGIN

# TYPE fish_payara_jeemongo_DepartmentResource_controller_counter_total counter
# HELP fish_payara_jeemongo_DepartmentResource_controller_counter_total
fish_payara_jeemongo_DepartmentResource_controller_counter_total{mp_scope="application"} 8
```

The @Metric annotation can be used to inject only metrics of types Meter, Timer, Counter, and Histogram (not covered in this guide). For situations where you want to have greater control of the metrics gathering, then you might want to inject the metrics manually and do the actual work of setting the requisite values/metrics.

Payara Specific Vendor Metric

One of the unique features of the comprehensive Payara MicroProfile implementation is the exposure of JMX Beans as part of the vendor metrics. As JMX precedes the MicroProfile Metrics spec, there are a lot of applications out there that have been implemented through the JMX API. Being able to expose such metrics as part of the vendor metrics makes Payara a perfect drop-in replacement runtime.

Within this feature, you can expose the name of the currently running Payara instance from which a given set of metrics originates. To achieve this, we first need to add the JMX metrics.xml descriptor to the ${PAYARA_HOME}/glassfish/domains/${DOMAIN_NAME}/config/ folder, where ${PAYARA_HOME} refers to the directory containing the Payara server bundle. This can also be copied to the folder using the COPY command when the application is packaged as a docker image. Within the metrics.xml file, create a metadata element with the name as ${instance} as shown in the example snippet below.

```
<config>
    <vendor>
        <metadata>
            <tags>
                <tag>
                    <name>operatingSystem</name>
                    <value>Linux</value>
                </tag>
            </tags>
            <name>requestcount.${instance)</name>
            <mbean>amx:type=web-request-mon, pp=/mon/server-mon
[${instance}],
                name=clusterjsp/server/requestcount#count
            </mbean>
            <type>gauge</type>
            <unit>none</unit>
            <displayName>Request count for hello-cluster</displayName>
        </metadata>
    </vendor>
</config>
```

The code declares a <metadata> element within the <vendor> element. This makes this metric available under the /metrics?scope=vendor path. We then optionally declare tags for the metrics within the <tags> element. We set the name of this metric as requestcount.${instance}. This will be automatically substituted at runtime with the configured name of the running Payara instance. For instance if the currently running instance is called prodEmea, the name of the metric will be vendor_requestcount_prodEmea 1. This metric can also be directly queried by its name.

Having the name of the currently running instance is a great enhancement to obtaining actionable metrics from applications deployed to Payara instances. Even though this example showed how to expose a single custom vendor metric, there is a lot more you can do with the Payara and MicroProfile Metrics. Take a look at the Metrics documentation on the Payara website for more.

Conclusion

The Eclipse MicroProfile Metrics API is a powerful and easy to use API to expose critical metrics about your microservices. Gathering insightful and actionable metrics about your applications is even more crucial in this era of cloud native application development and deployment.

With the Payara Platform fully implementing the latest MicroProfile specification, you are assured of a powerful platform on which to run your mission critical enterprise Java workload.

Should you need further support or info about using or transitioning your enterprise Java workload to the Payara Platform, please don't hesitate to get in touch with us. We would love to hear from you. You can also keep in touch with us on our social media platforms - Twitter, YouTube, GitHub.

This is also the latest guide in a series focusing on MicroProfile 6 – find the others here:

- Jakarta EE Application Health Check With MicroProfile Health
- Build Resilient Cloud Native Applications With MicroProfile Fault Tolerance
- Effortless Application Configuration with MicroProfile Config

sales@payara.fish

UK: +44 800 538 5490
Intl: +1 888 239 8941

www.payara.fish