User Guide

# How to Use Eclipse MicroProfile Metrics API with Payara Server

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

# Contents

## Introduction

The goal of this guide is to help you as a Java developer make the most of the Eclipse MicroProfile (EMP) Metrics API using the Payara Server. The guide starts by looking at what Eclipse MicroProfile is, the individual APIs that make up the EMP, and finally, takes a deeper look at the Metrics API. By the end of this guide, you will be able to integrate EMP into your application and learn how to build reliable and well functioning applications on the Payara Server.

## What Do I Need to Use This Guide?

You will need a copy of the latest Payara Server full stream distribution to follow along this guide. The sample code is built using Apache Maven and as such, any IDE that supports Maven is good enough. You should also have a minimum of Java SE 8 installed on your computer.

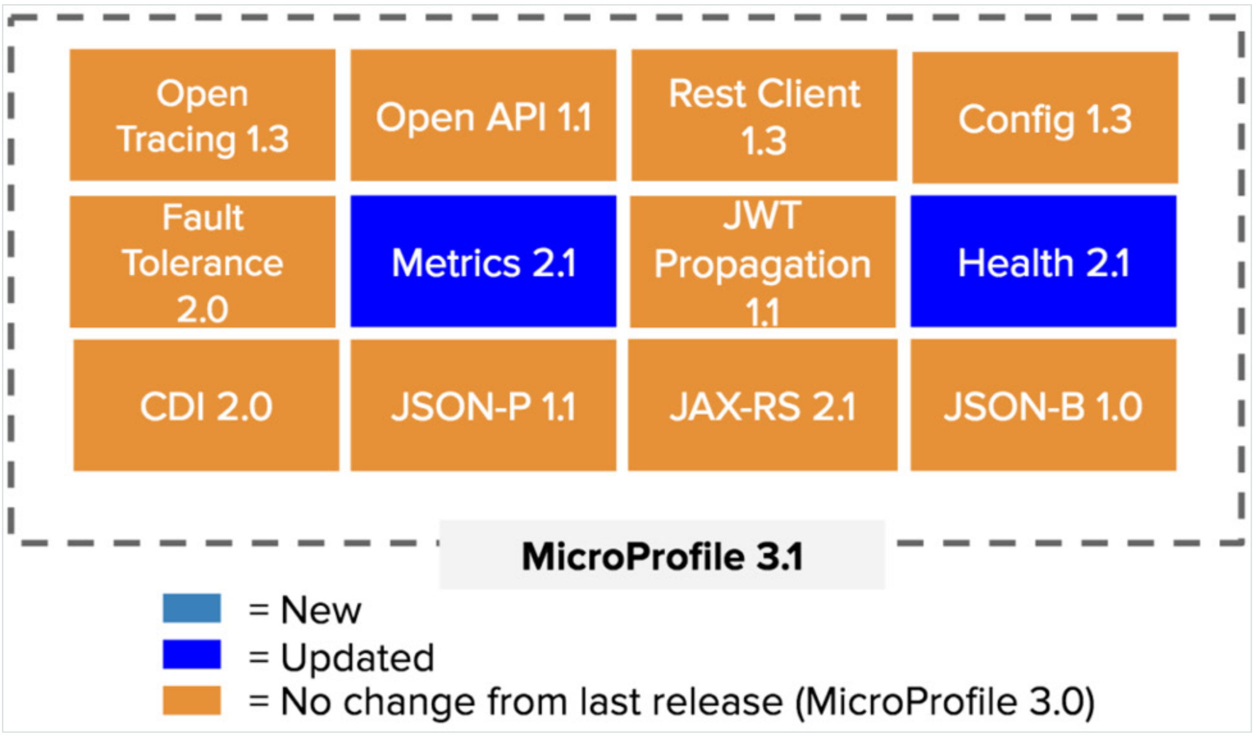
You can also follow along with the information in this guide with our video demo:

How to Use Eclipse MicroProfile Metrics API with Payara Server
Tutorial

## What is MicroProfile?

MicroProfile is an Eclipse Foundation led, community driven initiative, that consists of a collection of abstract specs that form a complete solution for developing cloud native, Java enterprise microservices. The goal is to create a set of APIs that removes you from their implementations so that you can create highly portable microservices across multiple vendors.

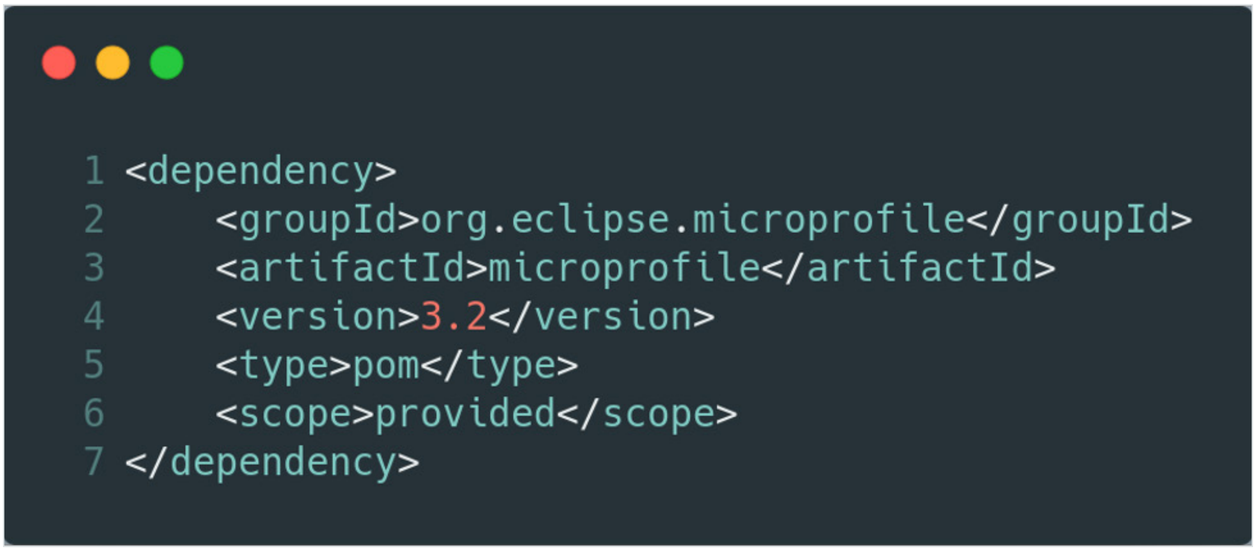
The current release is version 3.2 which reverts an inadvertent backward incompatible change introduced in Metrics 2.1 as part of MicroProfile 3.1. Therefore, MicroProfile 3.2 corrects Metrics, and re-introduces Health APIs. Like its previous version, MicroProfile 3.2 continues to align itself with Java EE 8 as the foundational programming model for the development of Java microservices and consists of 12 different specifications as shown below.



As abstract specifications, the various implementations are free to implement the base specs and add custom features on top. The Payara Server is one of the popular implementations of the MicroProfile spec and it adds quite a number of custom features on top of the base specs. This guide will walk you through the Payara Server implementation of the MicroProfile Metrics API.

## 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.



```xml
<dependency>
    <groupId>org.eclipse.microprofile</groupId>
    <artifactId>microprofile</artifactId>
    <version>3.2</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.

## 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.

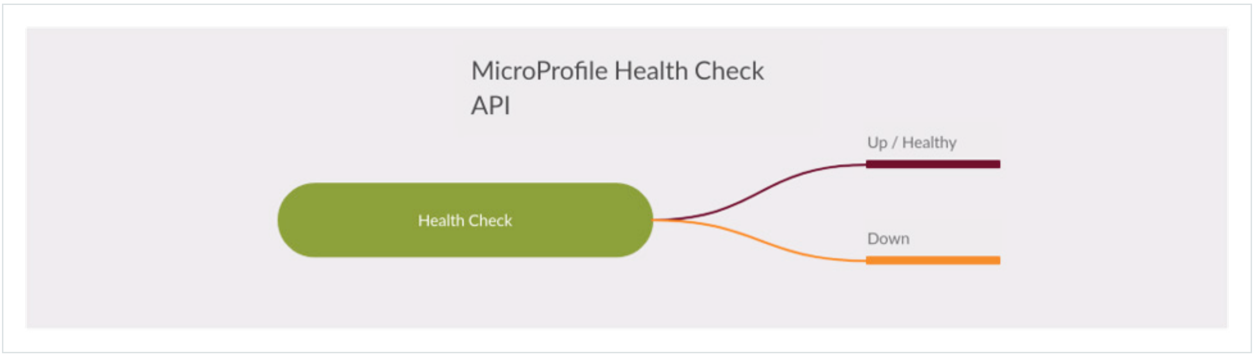
Typical metrics you would want to gather about your application could include how long a given method takes to execute, how many times a given method is invoked, how many times a given class is instantiated, and what the average concurrent access is to a given method per period, among other examples.

These are all metrics that need to be collected and monitored to ensure they meet set standards of an organization. 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.

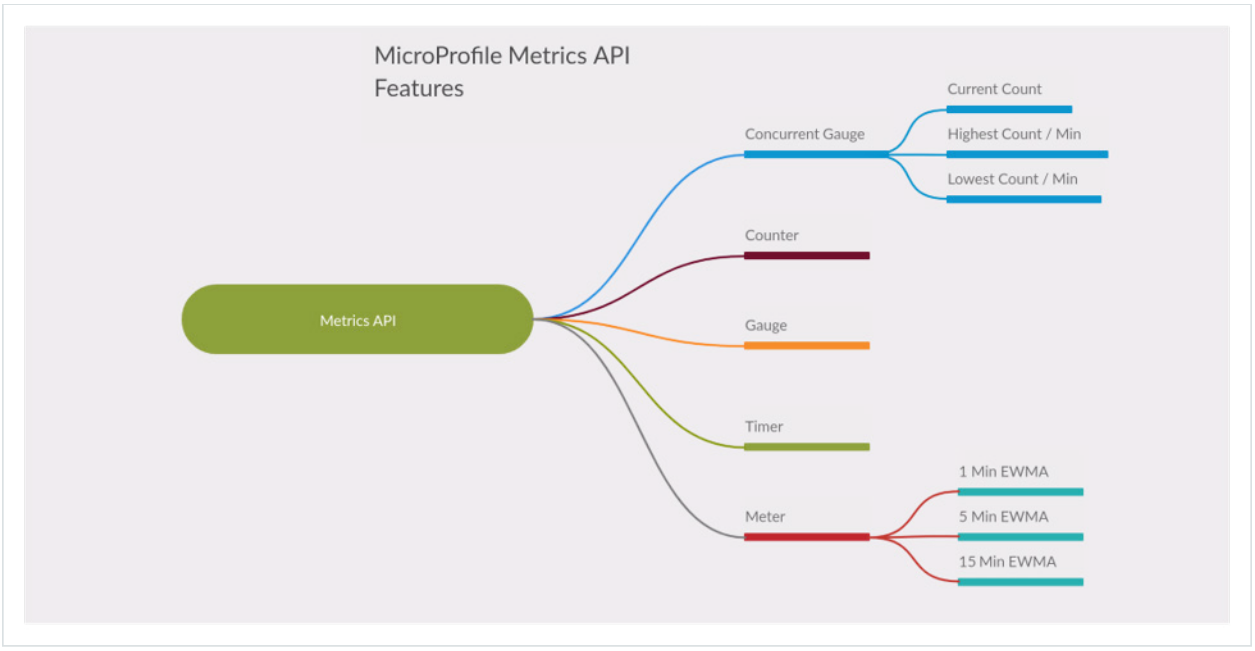
The Metrics API gives you a set of annotations and Java APIs that you can use to easily expose application metrics in the OpenMetrics format that can be consumed by systems monitoring and alerting toolkit like Prometheus. The Metrics API also supports exposing metrics in the JSON format.

## 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 microservices. 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. Contrast the above image with that of the Metrics API below.



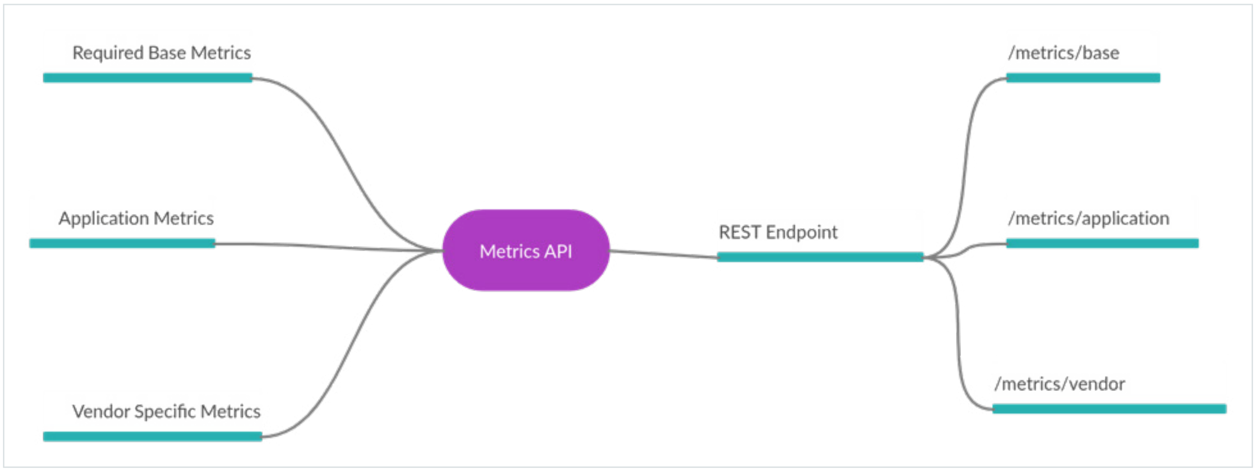
The MicroProfile Metrics API helps you monitor the extent to which an application is healthy, or not. For instance, you may have a method that takes 15 seconds to respond to a request. However, your expectation is that this particular method should take no more than 9 seconds to do so. In this case, 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:

- Concurrent gauge: an incrementally increasing or decreasing numeric value (e.g. number of parallel invocations of a method).
- 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.
- 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 (not shown in diagram).

## 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 all MicroProfile compliant servers must 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. Some metrics under the base scope are optional because they might be particular type or release of the underlying JVM or operating system. Examples of the optional base metrics are active thread pools and system load average.

### 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 examples.

### 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 (in the folder `${PAYARA_HOME}/glassfish/domains/${DOMAIN_NAME}/config/`).

## 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 to the /metrics resource returns metrics data in the Open Metrics text format. This format is an open standard supported by some metrics monitoring applications like Prometheus. The metrics endpoint can also return the metrics data in the JSON format if the HTTP Accept header best matches application/json. So in effect, the metrics HTTP endpoint returns

- JSON format - if the HTTP Accept header best matches application/json.
- OpenMetrics text format - if 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).

### Metrics Scopes Paths

The 3 metrics scopes - required base, application. and custom vendor metrics - all have their respective rest endpoints for accessing metrics specific to them. They are all relative to the base /metrics path. These are:

- /metrics/base for required base metrics
- /metrics/application for application specific metrics
- /metrics/vendor for custom vendor metrics

These sub-paths also do follow the metrics data format discussed above. Thus metrics for any of the scopes can be exposed as either JSON or the OpenMetrics text format, depending on the HTTP Accept header value, or the lack of it.

## 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 `displayName` - Optional. A human readable display name for metadata.
- String `description` - Optional. A description of the metric.
- String `unit` - Unit of the metric.
- boolean `reusable` - Denotes if a metric with a certain MetricID can be registered in more than one place.

These parameters determine the metadata for a given annotation.

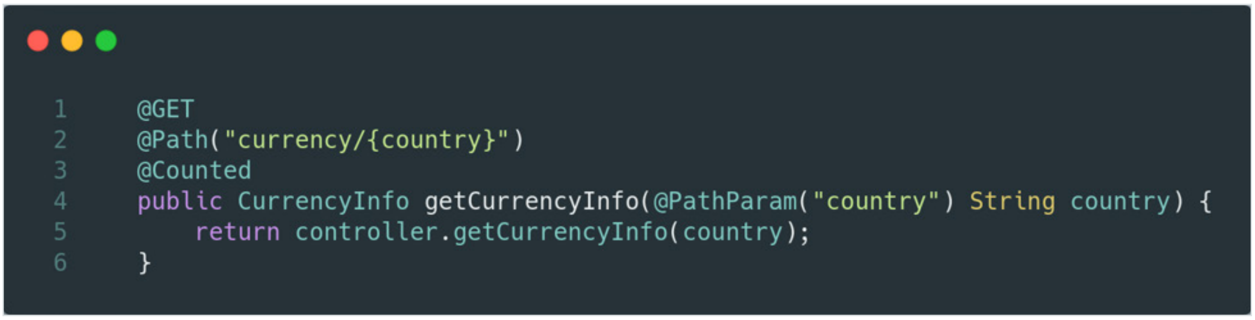
### The MetricsRegistry

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.

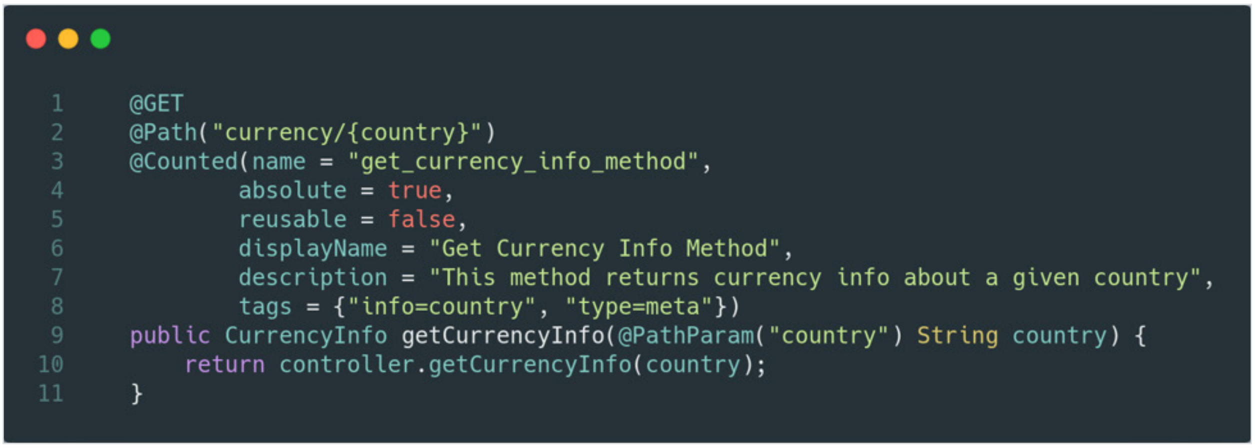
## @Counted

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 image below is annotated @Counted with no parameters.



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

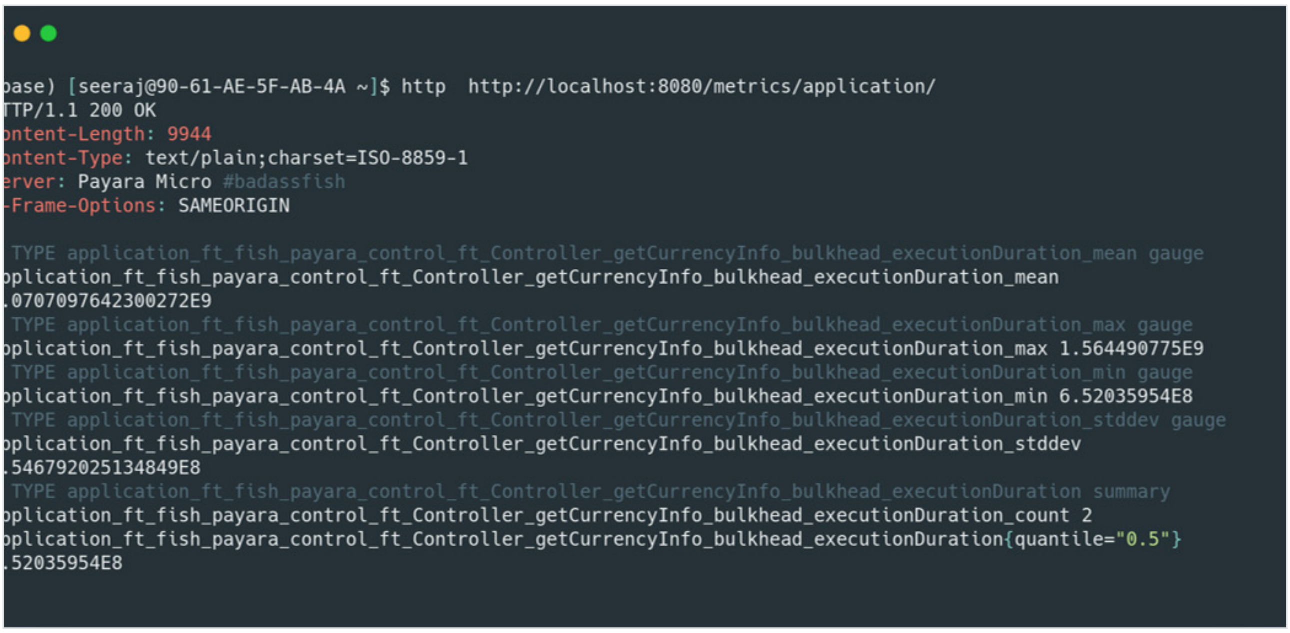
The `getCurrencyInfo` (Entire sample code is available for download from the Payara Web Site) method is annotated with @Counted, meaning for every invocation of this method, the counter will be incremented by 1. The name of this metric will be the fully qualified method name. Now let us add some custom metadata to the *@Counted* annotation.



```java
 1    @GET
 2    @Path("currency/{country}")
 3    @Counted(name = "get_currency_info_method",
 4            absolute = true,
 5            reusable = false,
 6            displayName = "Get Currency Info Method",
 7            description = "This method returns currency info about a given country",
 8            tags = {"info=country", "type=meta"})
 9    public CurrencyInfo getCurrencyInfo(@PathParam("country") String country) {
10        return controller.getCurrencyInfo(country);
11    }
```

In the above image, 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 on line 4. 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 reusable flag is set to false to prevent this metric from being used / registered in any other place. The reusable flag is false by default thus we could have omitted it and still attain the same effect. We also set the display name to a human readable string. The description has also been set to a brief text about the function of this metric. On line 8, we 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/application` endpoint when this code is deployed. By default, a GET request to this endpoint returns a text OpenMetrics format as shown below.

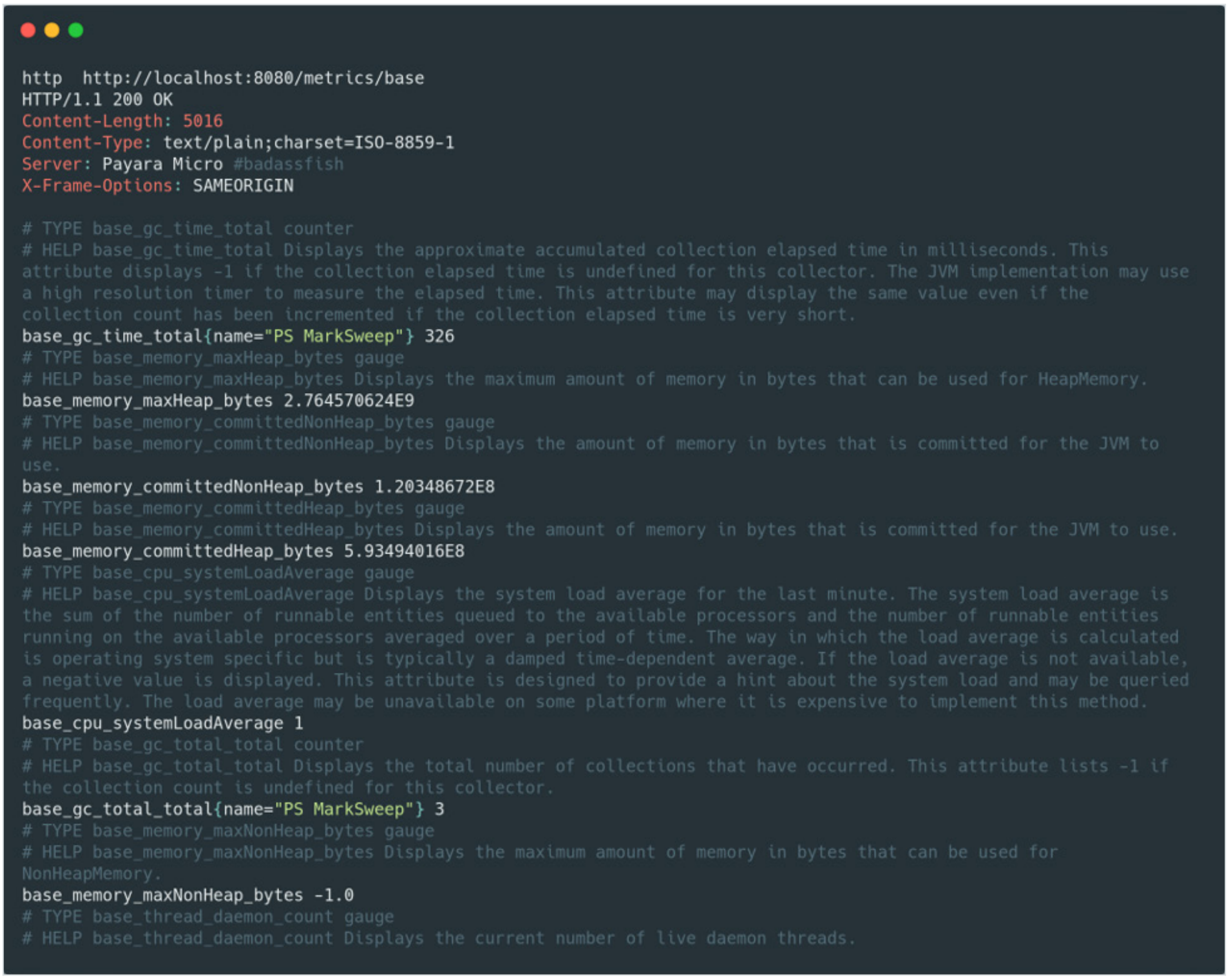


```
base) [seeraj@90-61-AE-5F-AB-4A ~]$ http  http://localhost:8080/metrics/application/
TTP/1.1 200 OK
ontent-Length: 9944
ontent-Type: text/plain;charset=ISO-8859-1
erver: Payara Micro #badassfish
-Frame-Options: SAMEORIGIN

 TYPE application_ft_fish_payara_control_ft_Controller_getCurrencyInfo_bulkhead_executionDuration_mean gauge
pplication_ft_fish_payara_control_ft_Controller_getCurrencyInfo_bulkhead_executionDuration_mean
.0707097642300272E9
 TYPE application_ft_fish_payara_control_ft_Controller_getCurrencyInfo_bulkhead_executionDuration_max gauge
pplication_ft_fish_payara_control_ft_Controller_getCurrencyInfo_bulkhead_executionDuration_max 1.564490775E9
 TYPE application_ft_fish_payara_control_ft_Controller_getCurrencyInfo_bulkhead_executionDuration_min gauge
pplication_ft_fish_payara_control_ft_Controller_getCurrencyInfo_bulkhead_executionDuration_min 6.52035954E8
 TYPE application_ft_fish_payara_control_ft_Controller_getCurrencyInfo_bulkhead_executionDuration_stddev gauge
pplication_ft_fish_payara_control_ft_Controller_getCurrencyInfo_bulkhead_executionDuration_stddev
.546792025134849E8
 TYPE application_ft_fish_payara_control_ft_Controller_getCurrencyInfo_bulkhead_executionDuration summary
pplication_ft_fish_payara_control_ft_Controller_getCurrencyInfo_bulkhead_executionDuration_count 2
pplication_ft_fish_payara_control_ft_Controller_getCurrencyInfo_bulkhead_executionDuration{quantile="0.5"}
.52035954E8
```

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", "type=meta"}       34`. This line refers to our *@Counted* annotated method. The number 34 means this method has been invoked 34 times in total.

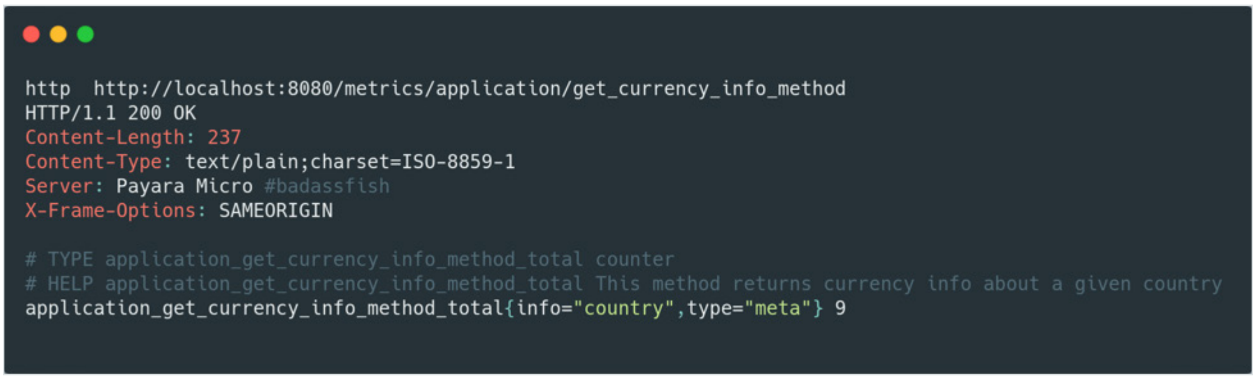
If you examine all the metrics in the above image, you can see they are all preceded by application. This is the way to identify that these metrics are from the application scope when metrics data is returned in text format. Similarly, a request to metrics/base returns metrics data prepended by base as shown below.

All the metrics you see in the image above except `application_get_currency_info_method_total{"info=country", "type=meta"}      34` are metrics from the MicroProfile Fault Tolerance API used in parts of the code. These are free metrics you get just by using the FT API in your code.



```
http  http://localhost:8080/metrics/base
HTTP/1.1 200 OK
Content-Length: 5016
Content-Type: text/plain;charset=ISO-8859-1
Server: Payara Micro #badassfish
X-Frame-Options: SAMEORIGIN

# TYPE base_gc_time_total counter
# HELP base_gc_time_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.
base_gc_time_total{name="PS MarkSweep"} 326
# TYPE base_memory_maxHeap_bytes gauge
# HELP base_memory_maxHeap_bytes Displays the maximum amount of memory in bytes that can be used for HeapMemory.
base_memory_maxHeap_bytes 2.764570624E9
# TYPE base_memory_committedNonHeap_bytes gauge
# HELP base_memory_committedNonHeap_bytes Displays the amount of memory in bytes that is committed for the JVM to
use.
base_memory_committedNonHeap_bytes 1.20348672E8
# TYPE base_memory_committedHeap_bytes gauge
# HELP base_memory_committedHeap_bytes Displays the amount of memory in bytes that is committed for the JVM to use.
base_memory_committedHeap_bytes 5.93494016E8
# TYPE base_cpu_systemLoadAverage gauge
# HELP base_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 platform where it is expensive to implement this method.
base_cpu_systemLoadAverage 1
# TYPE base_gc_total_total counter
# HELP base_gc_total_total Displays the total number of collections that have occurred. This attribute lists -1 if
the collection count is undefined for this collector.
base_gc_total_total{name="PS MarkSweep"} 3
# TYPE base_memory_maxNonHeap_bytes gauge
# HELP base_memory_maxNonHeap_bytes Displays the maximum amount of memory in bytes that can be used for
NonHeapMemory.
base_memory_maxNonHeap_bytes -1.0
# TYPE base_thread_daemon_count gauge
# HELP base_thread_daemon_count Displays the current number of live daemon threads.
```

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/application/get_currency_info_method` as shown below.



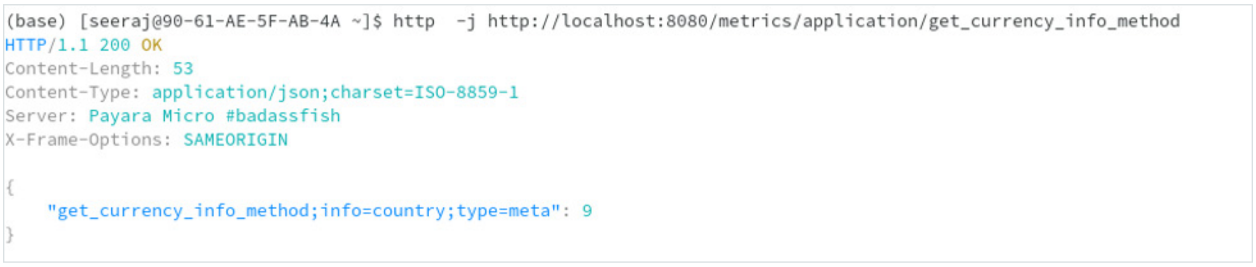
```
http  http://localhost:8080/metrics/application/get_currency_info_method
HTTP/1.1 200 OK
Content-Length: 237
Content-Type: text/plain;charset=ISO-8859-1
Server: Payara Micro #badassfish
X-Frame-Options: SAMEORIGIN

# TYPE application_get_currency_info_method_total counter
# HELP application_get_currency_info_method_total This method returns currency info about a given country
application_get_currency_info_method_total{info="country",type="meta"} 9
```

We could also get the same application scope metrics data back in JSON format by specifying a JSON Accept HTTP header value as shown below.

```
http -j  http://localhost:8080/metrics/application
HTTP/1.1 200 OK
Content-Length: 1934
Content-Type: application/json;charset=ISO-8859-1
Server: Payara Micro #badassfish
X-Frame-Options: SAMEORIGIN

{
    "fish.payara.boundary.ft.MicroProfileRestExamples.getCountryList": {
        "count": 0,
        "fifteenMinRate": 0.0,
        "fiveMinRate": 0.0,
        "meanRate": 0.0,
        "oneMinRate": 0.0
    },
    "fish.payara.boundary.ft.MicroProfileRestExamples.get_country_list": {
        "current": 0,
        "max": 0,
        "min": 0
    },
    "fish.payara.boundary.ft.MicroProfileRestExamples.get_country_list_timer": {
        "count;info=list;type=timer": 0,
        "fifteenMinRate;info=list;type=timer": 0.0,
        "fiveMinRate;info=list;type=timer": 0.0,
        "max;info=list;type=timer": 0,
        "mean;info=list;type=timer": 0.0,
        "meanRate;info=list;type=timer": 0.0,
        "min;info=list;type=timer": 0,
        "oneMinRate;info=list;type=timer": 0.0,
        "p50;info=list;type=timer": 0.0,
        "p75;info=list;type=timer": 0.0,
        "p95;info=list;type=timer": 0.0,
        "p98;info=list;type=timer": 0.0,
        "p999;info=list;type=timer": 0.0,
        "p99;info=list;type=timer": 0.0,
        "stddev;info=list;type=timer": 0.0
    },
    "fish.payara.control.ft.Controller.controller_counter": 9,
    "ft.fish.payara.control.ft.Controller.getCurrencyInfo.bulkhead.callsAccepted.total": 9,
    "ft.fish.payara.control.ft.Controller.getCurrencyInfo.bulkhead.concurrentExecutions": 0,
    "ft.fish.payara.control.ft.Controller.getCurrencyInfo.bulkhead.executionDuration": {
        "count": 9,
        "max": 1298788730,
        "mean": 628934916.6808774,
        "min": 559327298,
        "p50": 587462200.0,
        "p75": 598620965,
        "p95": 1298788730.0,
        "p98": 1298788730.0,
        "p99": 1298788730.0,
        "p999": 1298788730.0,
        "stddev": 160877831.81820384
    }
```

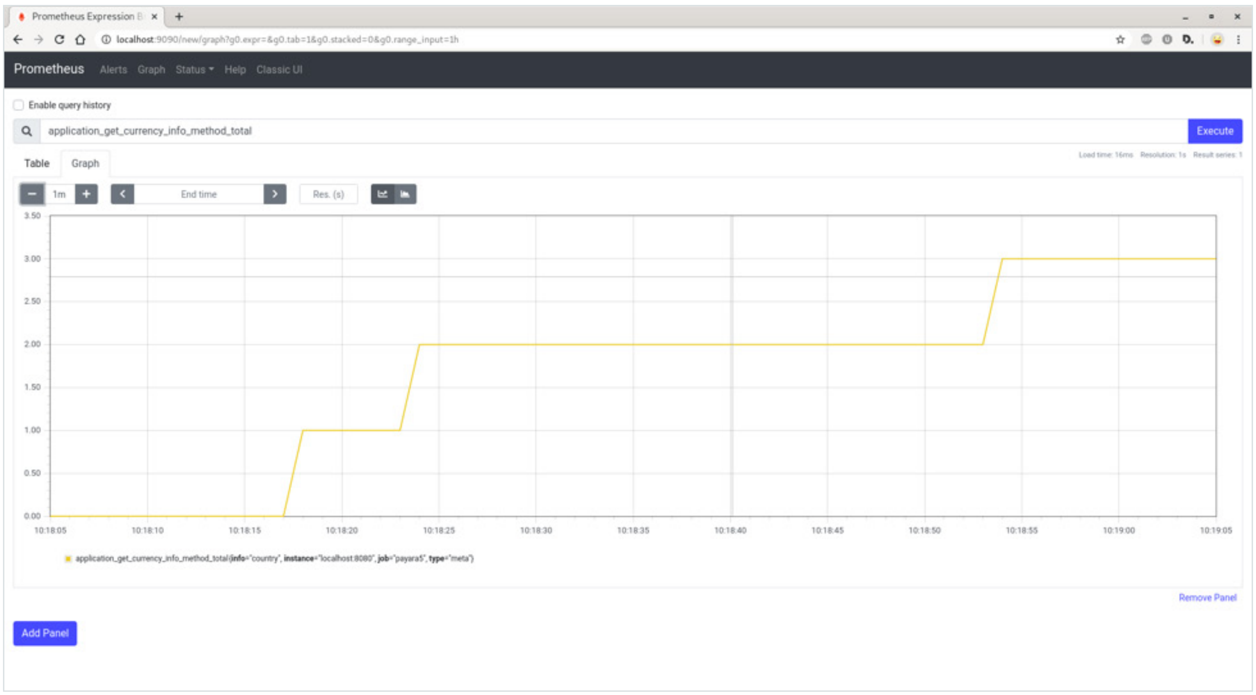
The last line in the image above shows the same 34 as being the number of times our `getCurrencyInfo` method has been invoked. The JSON data looks easy on the eyes and is very straight forward to read. As you can see, JSON format does not have the scope prepended because by default the scope of a metric acts as the key with the JSON object containing the data as value.

The metric for the annotated method can also be returned as JSON by making a request to `/metrics/application/application_get_currency_info_method_total`, specifying a JSON HTTP Accept header as shown below.



```
(base) [seeraj@90-61-AE-5F-AB-4A ~]$ http  -j http://localhost:8080/metrics/application/get_currency_info_method
HTTP/1.1 200 OK
Content-Length: 53
Content-Type: application/json;charset=ISO-8859-1
Server: Payara Micro #badassfish
X-Frame-Options: SAMEORIGIN

{
    "get_currency_info_method;info=country;type=meta": 9
}
```

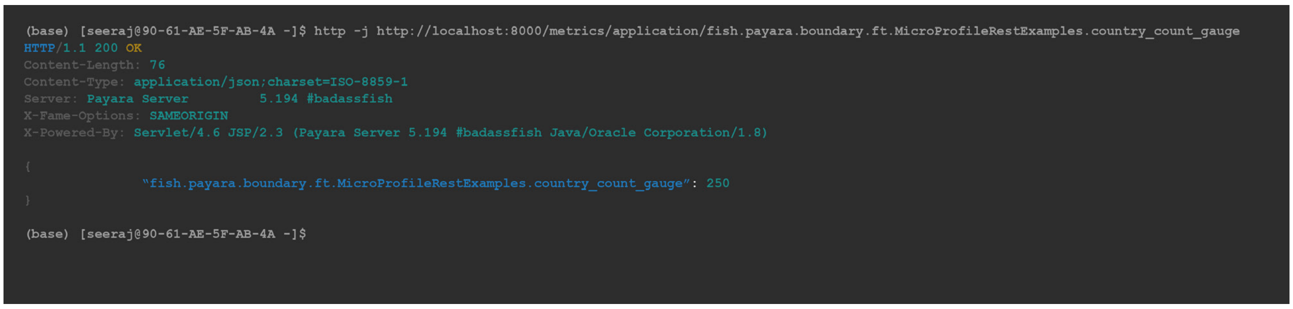
The metrics exposed by the Metrics API can be used as is with monitoring applications like Prometheus. For instance, the image below shows a graph of invocations of the `getCurrencyInfo` as mapped by Prometheus.



```
application_get_currency_info_method_total
```

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.



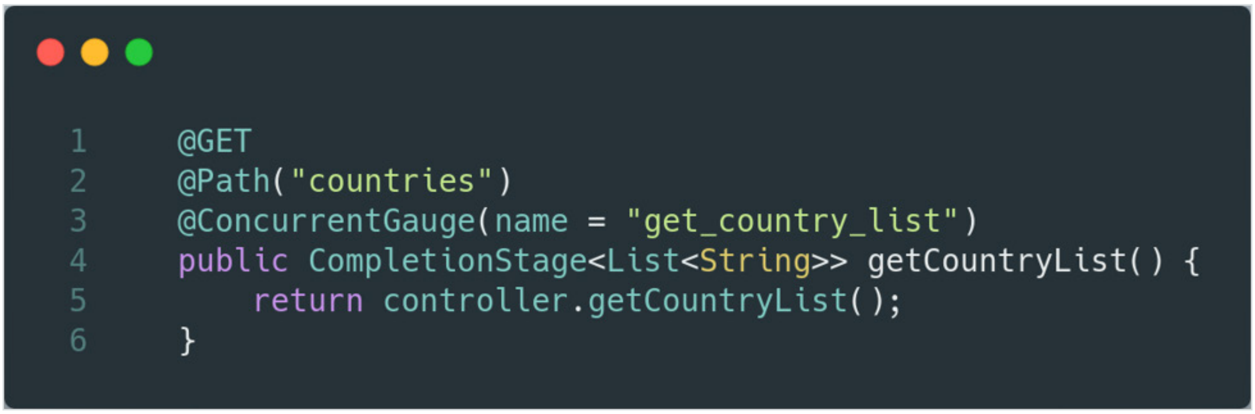
```
(base) [seeraj@90-61-AE-5F-AB-4A -]$ http -j http://localhost:8000/metrics/application/fish.payara.boundary.ft.MicroProfileRestExamples.country_count_gauge
HTTP/1.1 200 OK
Content-Length: 76
Content-Type: application/json;charset=ISO-8859-1
Server: Payara Server         5.194 #badassfish
X-Fame-Options: SAMEORIGIN
X-Powered-By: Servlet/4.6 JSP/2.3 (Payara Server 5.194 #badassfish Java/Oracle Corporation/1.8)

{
               "fish.payara.boundary.ft.MicroProfileRestExamples.country_count_gauge": 250
}

(base) [seeraj@90-61-AE-5F-AB-4A -]$
```

## @ConcurrentGauge

This annotation is used to gauge concurrent invocations of an annotated artefact. *@ConcurrentGauge* can be used on a class, constructor or method. This metric works by incrementing the parallel count by 1 when an annotated artefact is entered and decremented by 1 when exited. The aim of this annotation is to gauge the number of concurrent invocations of the annotated artefact.

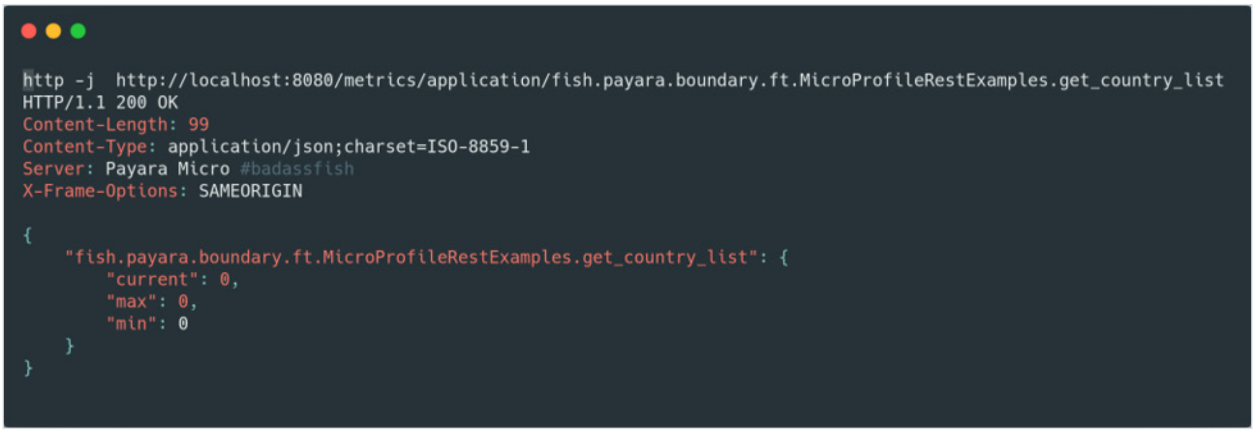
When used on a class, an parallel invocation counter for the current, the previous minute maximum and previous minute minimum are created and registered in the metric registry for each constructor and method of the class using the name and absolute field values of the annotation. When used on a method, metric gauges for the current, previous minute maximum and previous minute minimum are created and registered in the metric registry. When used on a constructor, the same gauges for that constructor and registered.

In the code snippet below, we annotate the method `getCountryList` with the *@ConcurrentGauge*, setting the name as `get_country_list`, leaving the absolute field with the default false.



```java
@GET
@Path("countries")
@ConcurrentGauge(name = "get_country_list")
public CompletionStage<List<String>> getCountryList() {
    return controller.getCountryList();
}
```

This makes this metric accessible from the REST endpoint `/metrics/application/fish.payara.boundary.ft.MicroProfileRestExamples.get_country_list` as shown below.

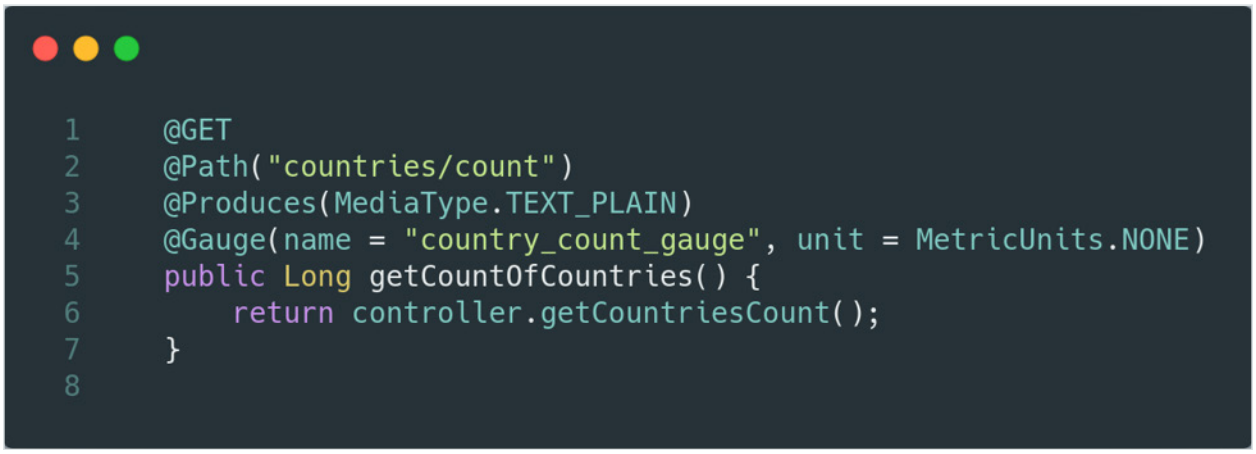


```
http -j  http://localhost:8080/metrics/application/fish.payara.boundary.ft.MicroProfileRestExamples.get_country_list
HTTP/1.1 200 OK
Content-Length: 99
Content-Type: application/json;charset=ISO-8859-1
Server: Payara Micro #badassfish
X-Frame-Options: SAMEORIGIN

{
    "fish.payara.boundary.ft.MicroProfileRestExamples.get_country_list": {
        "current": 0,
        "max": 0,
        "min": 0
    }
}
```

## @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. Unlike the metrics you have seen so far, *@Gauge* can be used only on a method, and the value must be supplied by the application.

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



```
1    @GET
2    @Path("countries/count")
3    @Produces(MediaType.TEXT_PLAIN)
4    @Gauge(name = "country_count_gauge", unit = MetricUnits.NONE)
5    public Long getCountOfCountries() {
6        return controller.getCountriesCount();
7    }
8
```

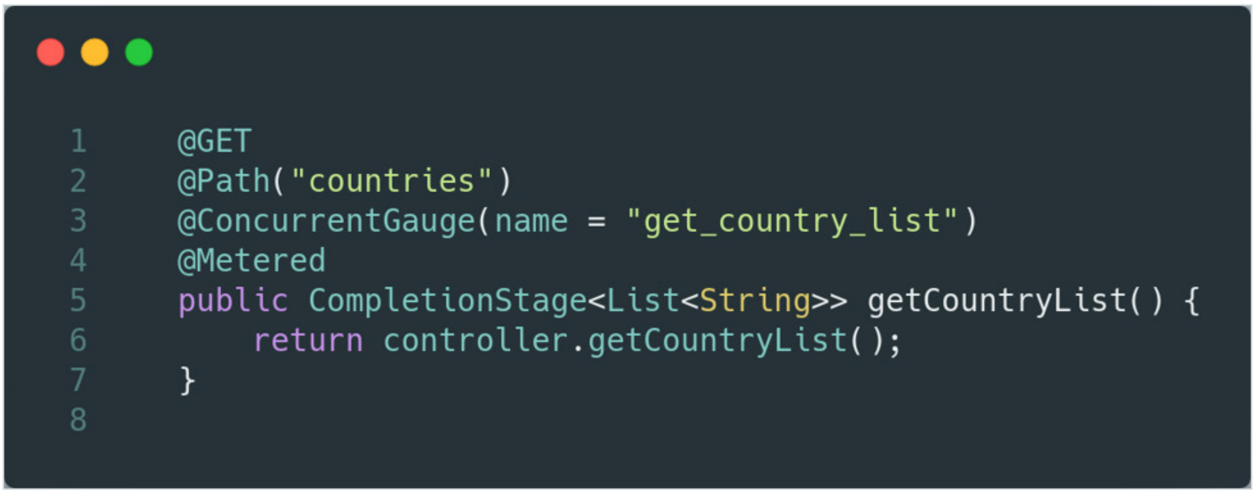
Line 4 annotates the method with *@Gauge*, setting the name to `count_count_gauge`. The metric unit for *@Gauge* is mandatory, so we set it to `MetricUnits.NONE` which defaults integer count. This unit lends itself better to the kind of metric we are returning as compared to the all the available ones from the `MetricsUnits` class.

A request for the metrics of the @Gauge annotated method through the REST endpoint returns the total count returned from the method as shown below.

## @Metered

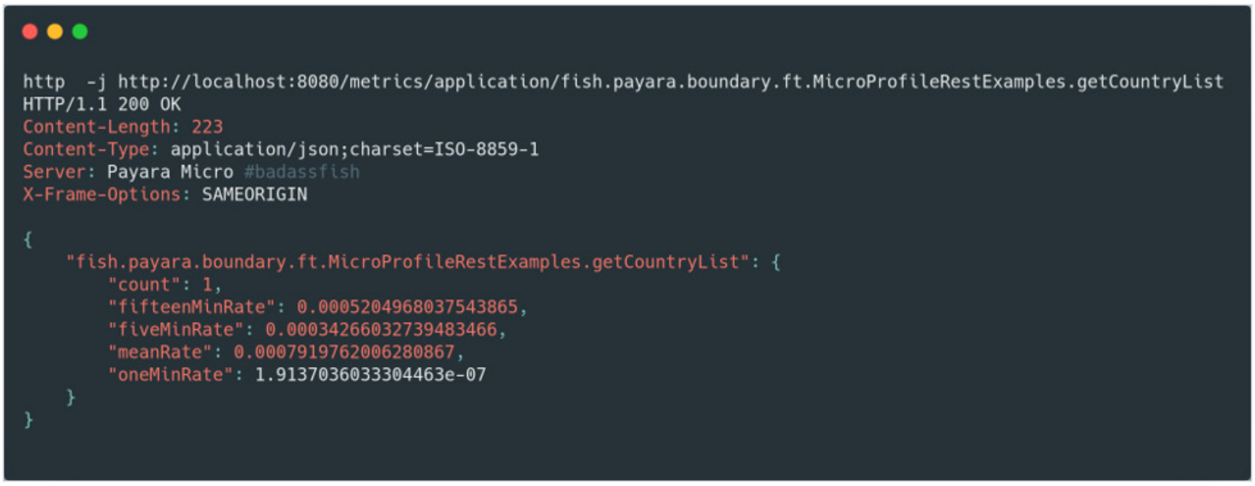
The *@Metered* annotation is a metric that tracks the frequency of invocations of the annotated artefact. It tracks the mean throughput and one-, five-, and fifteen-minute exponentially-weighted moving average throughput. The default MetricUnit is seconds. *@Metered* can be used on a class, method or constructor.

In our sample code, we want to track the frequency of invocations of the `getCountryList` method. As shown below, it is annotated with the *@Metered* metric annotation.



```
1    @GET
2    @Path("countries")
3    @ConcurrentGauge(name = "get_country_list")
4    @Metered
5    public CompletionStage<List<String>> getCountryList() {
6        return controller.getCountryList();
7    }
8
```

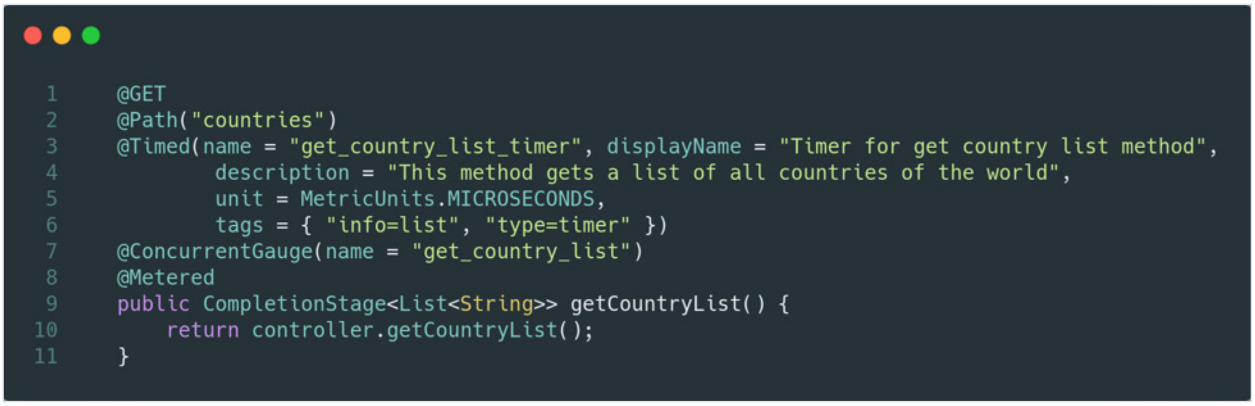
Line 4 of the above code shows the use of *@Metered* without any parameter. Because no name was specified, the metered metrics will be available through the fully qualified method name. An invocation to the REST `endpoint /metrics/application/fish.payara.boundary.ft.MicroProfileRestExamples.getCountryList` returns the metered metrics as shown below.



```
http  -j http://localhost:8080/metrics/application/fish.payara.boundary.ft.MicroProfileRestExamples.getCountryList
HTTP/1.1 200 OK
Content-Length: 223
Content-Type: application/json;charset=ISO-8859-1
Server: Payara Micro #badassfish
X-Frame-Options: SAMEORIGIN

{
    "fish.payara.boundary.ft.MicroProfileRestExamples.getCountryList": {
        "count": 1,
        "fifteenMinRate": 0.0005204968037543865,
        "fiveMinRate": 0.00034266032739483466,
        "meanRate": 0.0007919762006280867,
        "oneMinRate": 1.9137036033304463e-07
    }
}
```

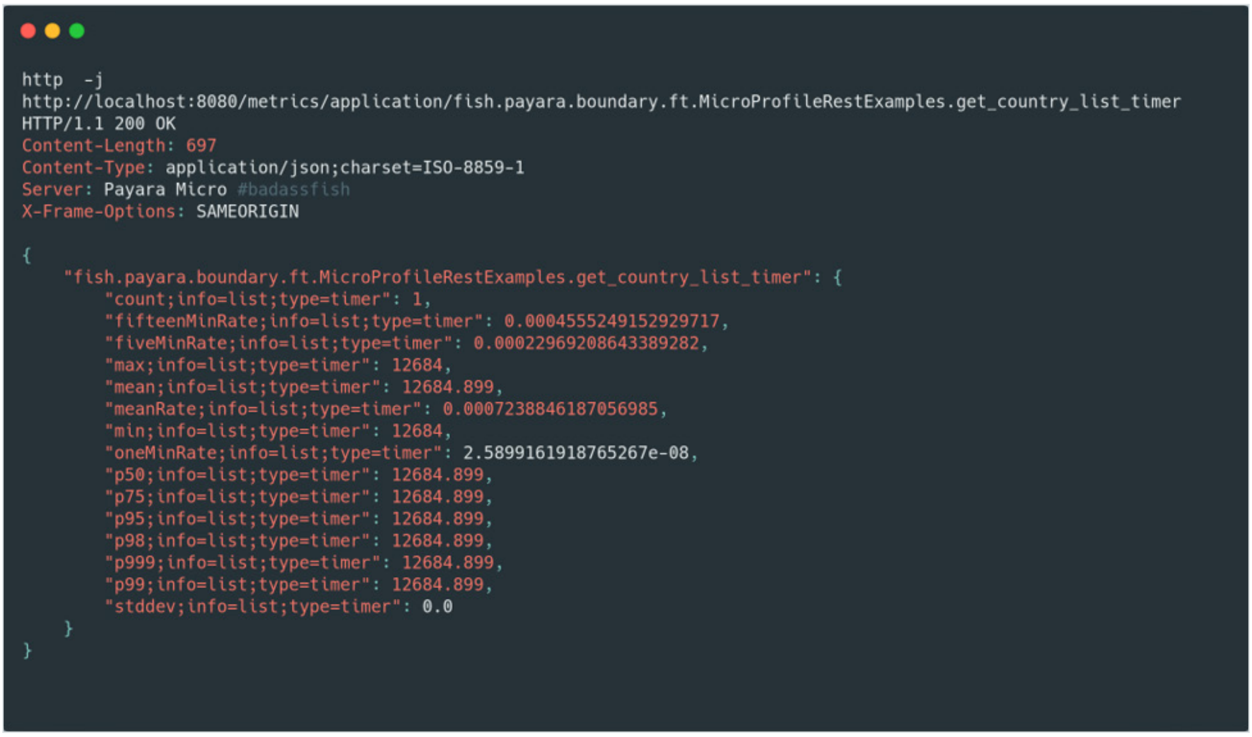
## @Timed

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.



```
 1    @GET
 2    @Path("countries")
 3    @Timed(name = "get_country_list_timer", displayName = "Timer for get country list method",
 4            description = "This method gets a list of all countries of the world",
 5            unit = MetricUnits.MICROSECONDS,
 6            tags = { "info=list", "type=timer" })
 7    @ConcurrentGauge(name = "get_country_list")
 8    @Metered
 9    public CompletionStage<List<String>> getCountryList() {
10        return controller.getCountryList();
11    }
```

Line 3 of the above code annotates method `getCountryList` with *@Timed* annotation. The name of the metric is set to `get_country_list_timer`. The display name, 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.

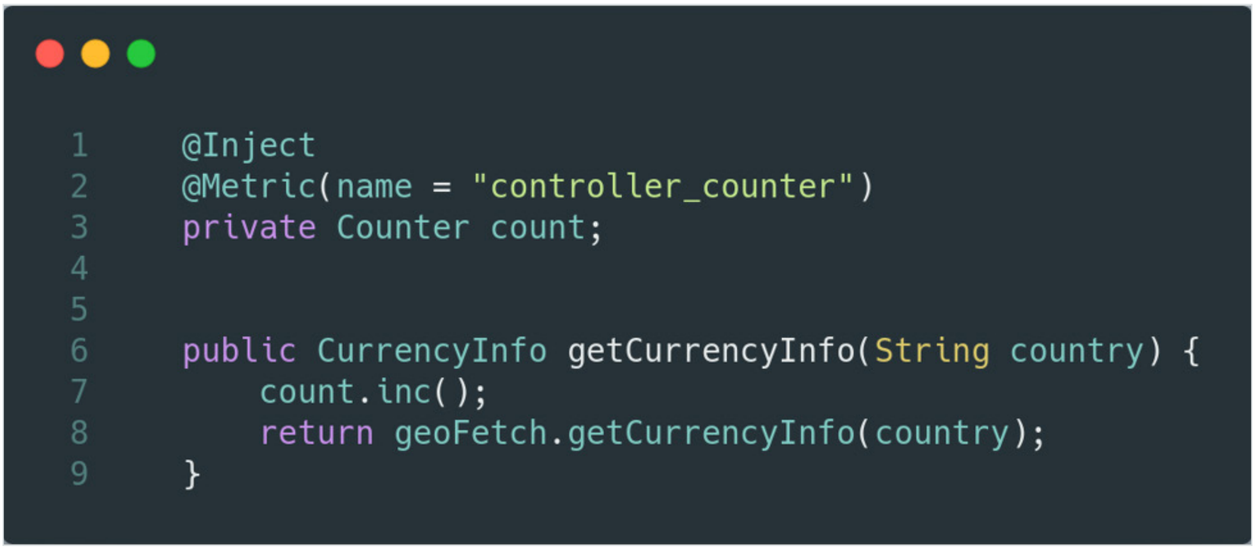


```
http  -j
http://localhost:8080/metrics/application/fish.payara.boundary.ft.MicroProfileRestExamples.get_country_list_timer
HTTP/1.1 200 OK
Content-Length: 697
Content-Type: application/json;charset=ISO-8859-1
Server: Payara Micro #badassfish
X-Frame-Options: SAMEORIGIN

{
    "fish.payara.boundary.ft.MicroProfileRestExamples.get_country_list_timer": {
        "count;info=list;type=timer": 1,
        "fifteenMinRate;info=list;type=timer": 0.0004555249152929717,
        "fiveMinRate;info=list;type=timer": 0.00022969208643389282,
        "max;info=list;type=timer": 12684,
        "mean;info=list;type=timer": 12684.899,
        "meanRate;info=list;type=timer": 0.0007238846187056985,
        "min;info=list;type=timer": 12684,
        "oneMinRate;info=list;type=timer": 2.5899161918765267e-08,
        "p50;info=list;type=timer": 12684.899,
        "p75;info=list;type=timer": 12684.899,
        "p95;info=list;type=timer": 12684.899,
        "p98;info=list;type=timer": 12684.899,
        "p999;info=list;type=timer": 12684.899,
        "p99;info=list;type=timer": 12684.899,
        "stddev;info=list;type=timer": 0.0
    }
}
```

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

## Injecting Metrics

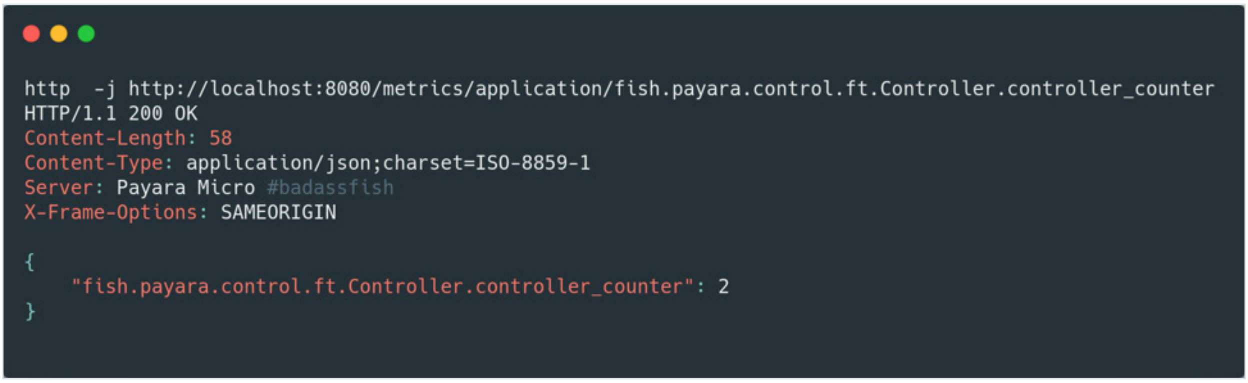
The metrics you have seen so far have all been used in the form of annotations. The Metrics API also supports injecting metrics types into fields, methods or parameters and then manually use those metrics using the *@Metric* annotation. For instance, in our sample code, we would want to inject a counter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



```
1    @Inject
2    @Metric(name = "controller_counter")
3    private Counter count;
4
5
6    public CurrencyInfo getCurrencyInfo(String country) {
7        count.inc();
8        return geoFetch.getCurrencyInfo(country);
9    }
```

Lines 1 and 2 of the above code snippet from `Controller.java` class annotate 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. On line 7, the inc() method on the count object is invoked to increment the counter by 1 for every invocation of the method `getCurrencyInfo`.

When we used the *@Counted* annotation, we were able to pass some metadata to be associated with the counter. The *@Metric* annotation used on line 2 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/application/fish.payara.control.ft.Controller.controller_counter` as shown below.



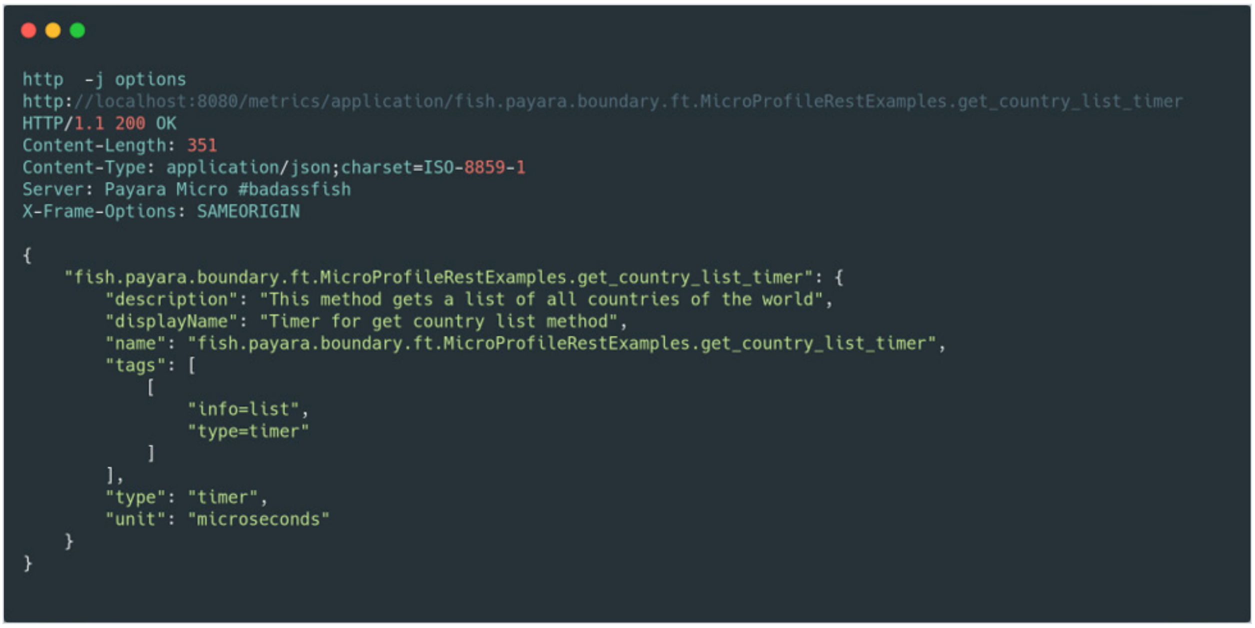
```
http  -j http://localhost:8080/metrics/application/fish.payara.control.ft.Controller.controller_counter
HTTP/1.1 200 OK
Content-Length: 58
Content-Type: application/json;charset=ISO-8859-1
Server: Payara Micro #badassfish
X-Frame-Options: SAMEORIGIN

{
    "fish.payara.control.ft.Controller.controller_counter": 2
}
```

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.

## Metadata

Metrics metadata are all the information that help describe a given metric. Metrics metadata can be retrieved from the REST endpoint for a specific scope by making a HTTP OPTIONS request to the REST endpoint for the scope. A OPTIONS request to any of the scopes or even the base /metrics REST endpoint returns all registered metrics for the scope.

For instance, to obtain the metadata associated with the *@Timed* metric as used on the `getCountryList` method, we make a HTTP OPTIONS request to the endpoint `/metrics/application/fish.payara.boundary.ft.MicroProfileRestExamples.get_country_list_timer` as shown below.



```
http  -j options
http://localhost:8080/metrics/application/fish.payara.boundary.ft.MicroProfileRestExamples.get_country_list_timer
HTTP/1.1 200 OK
Content-Length: 351
Content-Type: application/json;charset=ISO-8859-1
Server: Payara Micro #badassfish
X-Frame-Options: SAMEORIGIN

{
    "fish.payara.boundary.ft.MicroProfileRestExamples.get_country_list_timer": {
        "description": "This method gets a list of all countries of the world",
        "displayName": "Timer for get country list method",
        "name": "fish.payara.boundary.ft.MicroProfileRestExamples.get_country_list_timer",
        "tags": [
            [
                "info=list",
                "type=timer"
            ]
        ],
        "type": "timer",
        "unit": "microseconds"
    }
}
```

Requests for metrics metadata are by default, returned in the JSON format as shown above. Everything set as part of the metadata for the @Timed annotation has been returned in the metadata request.

## 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 Server 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 Server, 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.

**sales@payara.fish** **+44 207 754 0481** **www.payara.fish**