Quarkus - MicroProfile Metrics

This guide demonstrates how your Quarkus application can utilize the MicroProfile Metrics specification through the SmallRye Metrics extension.

MicroProfile Metrics allows applications to gather various metrics and statistics that provide insights into what is happening inside the application.

The metrics can be read remotely using JSON format or the OpenMetrics format, so that they can be processed by additional tools such as Prometheus, and stored for analysis and visualisation.

Apart from application-specific metrics, which are described in this guide, you may also utilize built-in metrics exposed by various Quarkus extensions. These are described in the guide for each particular extension that supports built-in metrics.

Prerequisites

To complete this guide, you need:

- less than 15 minutes

- an IDE

- JDK 1.8+ installed with `JAVA_HOME` configured appropriately

- Apache Maven 3.5.3+

Architecture

In this example, we build a very simple microservice which offers one REST endpoint and that endpoint serves for determining whether a number is prime. The implementation class is annotated with some metric annotations so that while responding to user's requests, some metrics are gathered. The meaning of each metric will be explained later.

Solution

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can go right to the completed example.

Clone the Git repository: `git clone` `https://github.com/quarkusio/quarkus-quickstarts.git`, or download an archive.

The solution is located in the `microprofile-metrics-quickstart` directory.

Creating the Maven Project

First, we need a new project. Create a new project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.3.0.CR1:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=microprofile-metrics-quickstart \
    -Dextensions="metrics"
cd microprofile-metrics-quickstart
```

This command generates a Maven project, importing the `smallrye-metrics` extension which is an implementation of the MicroProfile Metrics specification used in Quarkus.

Writing the application

The application consists of a single class that implements an algorithm for checking whether a number is prime. This algorithm is exposed over a REST interface. Additionally, we need a few annotations to make sure that our desired metrics are calculated over time and can be exported for manual analysis or processing by additional tooling.

The metrics that we will gather are these:

- `performedChecks`: A counter which is increased by one each time the user asks about a number.

- `highestPrimeNumberSoFar`: This is a gauge that stores the highest number that was asked about by the user and which was determined to be prime.

- `checksTimer`: This is a timer, therefore a compound metric that benchmarks how much time the primality tests take. We will explain that one in more details later.

The full source code looks like this:

```java
package org.acme.microprofile.metrics;

import org.eclipse.microprofile.metrics.MetricUnits;
import org.eclipse.microprofile.metrics.annotation.Counted;
import org.eclipse.microprofile.metrics.annotation.Gauge;
import org.eclipse.microprofile.metrics.annotation.Timed;
import org.jboss.resteasy.annotations.jaxrs.PathParam;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@Path("/")
public class PrimeNumberChecker {

    private long highestPrimeNumberSoFar = 2;

    @GET
    @Path("/{number}")
    @Produces("text/plain")
```

```java
    @Counted(name = "performedChecks", description = "How many
primality checks have been performed.")
    @Timed(name = "checksTimer", description = "A measure of how
long it takes to perform the primality test.", unit = MetricUnits
.MILLISECONDS)
    public String checkIfPrime(@PathParam long number) {
        if (number < 1) {
            return "Only natural numbers can be prime numbers.";
        }
        if (number == 1) {
            return "1 is not prime.";
        }
        if (number == 2) {
                    return "2 is prime.";
        }
        if (number % 2 == 0) {
            return number + " is not prime, it is divisible by 2.";
        }
        for (int i = 3; i < Math.floor(Math.sqrt(number)) + 1; i =
i + 2) {
            if (number % i == 0) {
                return number + " is not prime, is divisible by " +
i + ".";
            }
        }
        if (number > highestPrimeNumberSoFar) {
            highestPrimeNumberSoFar = number;
        }
        return number + " is prime.";
    }

    @Gauge(name = "highestPrimeNumberSoFar", unit = MetricUnits
.NONE, description = "Highest prime number so far.")
    public Long highestPrimeNumberSoFar() {
        return highestPrimeNumberSoFar;
    }

}
```

Running and using the application

To run the microservice in dev mode, use `./mvnw clean compile quarkus:dev`

Generate some values for the metrics

First, ask the endpoint whether some numbers are prime numbers.

```
curl localhost:8080/350
```

The application will respond that 350 is not a prime number because it can be divided by 2.

Now for some large prime number so that the test takes a bit more time:

```
curl localhost:8080/629521085409773
```

The application will respond that 629521085409773 is a prime number. If you want, try some more calls with numbers of your choice.

Review the generated metrics

To view the metrics, execute `curl -H"Accept: application/json" localhost:8080/metrics/application` You will receive a response such as:

```
{
  "org.acme.microprofile.metrics.PrimeNumberChecker.checksTimer" :
{
    "p50": 217.231273,
    "p75": 217.231273,
    "p95": 217.231273,
    "p98": 217.231273,
    "p99": 217.231273,
    "p999": 217.231273,
    "min": 0.58961,
    "mean": 112.15909190834819,
    "max": 217.231273,
    "stddev": 108.2721053982776,
    "count": 2,
    "meanRate": 0.04943519091742238,
    "oneMinRate": 0.2232140583080189,
    "fiveMinRate": 0.3559527083952095,
    "fifteenMinRate": 0.38474303050928976
  },

 "org.acme.microprofile.metrics.PrimeNumberChecker.performedChecks"
 : 2,

 "org.acme.microprofile.metrics.PrimeNumberChecker.highestPrimeNumbe
 rSoFar" : 629521085409773
 }
```

Let's explain the meaning of each metric:

- `performedChecks`: A counter which is increased by one each time the user asks about a number.

- `highestPrimeNumberSoFar`: This is a gauge that stores the highest number that was asked about by the user and which was determined to be prime.

- `checksTimer`: This is a timer, therefore a compound metric that benchmarks how much time the primality tests take. All durations are measured in milliseconds. It consists of these values:

  - `min`: The shortest duration it took to perform a primality test, probably it was performed for a small number.

  - `max`: The longest duration, probably it was with a large prime number.

  - `mean`: The mean value of the measured durations.

  - `stddev`: The standard deviation.

  - `count`: The number of observations (so it will be the same value as `performedChecks`).

  - `p50, p75, p95, p99, p999`: Percentiles of the durations. For example the value in `p95` means that 95 % of the measurements were faster than this duration.

  - `meanRate, oneMinRate, fiveMinRate, fifteenMinRate`: Mean throughput and one-, five-, and fifteen-minute exponentially-weighted moving average throughput.

If you prefer an OpenMetrics export rather than the JSON format, remove the `-H"Accept: application/json"` argument from your command line.

Configuration Reference

🔒 Configuration property fixed at build time - All other configuration properties are overridable at runtime

Configuration property	Type	Default
🔒 `quarkus.smallrye-metrics.path` The path to the metrics handler.	string	`/metrics`
🔒 `quarkus.smallrye-metrics.extensions.enabled` Whether or not metrics published by Quarkus extensions should be enabled.	boolean	`true`
🔒 `quarkus.smallrye-metrics.micrometer.compatibility` Apply Micrometer compatibility mode, where instead of regular 'base' and 'vendor' metrics, Quarkus exposes the same 'jvm' metrics that Micrometer does. Application metrics are unaffected by this mode. The use case is to facilitate migration from Micrometer-based metrics, because original dashboards for JVM metrics will continue working without having to rewrite them.	boolean	`false`