Red Hat build of Quarkus 1.11

Collecting metrics in your Quarkus applications

Last Updated: 2021-08-23

Legal Notice

Abstract

Learn how to collect metrics in Quarkus using Micrometer and Prometheus.

Table of Contents

PREFACE

As an application developer, you can collect metrics using Micrometer to improve the performance of your Quarkus application.

Prerequisites

- Have OpenJDK (JDK) 11 installed and the location of the Java SDK specified by the **JAVA_HOME** environment variable.
    - Log in to the Red Hat Customer Portal to download Red Hat build of Open JDK from the Software Downloads page.
- Have Apache Maven 3.6.2 or higher installed. Maven is available from the Apache Maven Project website.

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your feedback on our technical content and encourage you to tell us what you think. If you'd like to add comments, provide insights, correct a typo, or even ask a question, you can do so directly in the documentation.

NOTE

You must have a Red Hat account and be logged in to the customer portal.

To submit documentation feedback from the customer portal, do the following:

1. Select the **Multi-page HTML** format.
2. Click the **Feedback** button at the top-right of the document.
3. Highlight the section of text where you want to provide feedback.
4. Click the **Add Feedback** dialog next to your highlighted text.
5. Enter your feedback in the text box on the right of the page and then click **Submit**.

We automatically create a tracking issue each time you submit feedback. Open the link that is displayed after you click **Submit** and start watching the issue or add more comments.

Thank you for the valuable feedback.

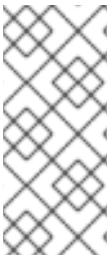
MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message.

CHAPTER 1. METRICS COLLECTION FOR YOUR QUARKUS APPLICATIONS

Metrics are quantitative measurements of specific aspects of an application that are used to observe trends and behavior. Individual measurements are collected at regular intervals with each observed numerical value uniquely identified by a string key and additional (optional) tags or labels.

These key-value pairs are then appended to a time series: a sequence of data points indexed over time. Capturing and analyzing metrics can help you identify potential issues and anomalies before they escalate and cause more serious problems.



NOTE

Metrics cannot be used for diagnostics or problem determination. Visualization tools aggregate individual measurements to provide visualizations of trends. The incident-specific context you need to identify the cause of an observed issue will not be found in aggregated metrics data; you need more detailed trace or log data for problem determination or root cause analysis.

You can collect runtime and application metrics using either the Micrometer library or SmallRye Metrics specifications:

- Micrometer provides a simple facade over the instrumentation clients for well-known monitoring systems. Quarkus pairs Micrometer with Prometheus to help you monitor and manage your application.
- SmallRye Metrics is an implementation of the MicroProfile Metrics specification that provides a Prometheus-compatible metrics endpoint.

The Micrometer extension is the recommended approach for gathering application and runtime metrics in Quarkus and provides the following:

- Dimensional metrics - vendor-neutral interfaces for timers, gauges, counters, distribution summaries, and long task timers with dimensional analysis that, when paired with a dimensional monitoring system, allows for efficient access to a particular named metric with the ability to drill down across its dimensions.
- Pre-configured bindings - out-of-the-box instrumentation of caches, the class loader, garbage collection, processor utilization, thread pools, and HTTP traffic. Other extensions, like **hibernate-orm** and **mongodb-client**, provide additional bindings automatically when enabled.

CHAPTER 2. EXPOSING METRICS IN YOUR QUARKUS APPLICATIONS

Enable metrics in Quarkus 1.11 using the **micrometer** extension. After you enable it, real-time values of all metrics collected by the **micrometer** extension are viewed using the **/q/metrics** endpoint. By default, this endpoint only responds in plain text.

Procedure

1. Add the **quarkus-micrometer-registry-prometheus** extension as a dependency to your application:

```
./mvnw quarkus:add-extension -Dextensions="io.quarkus:quarkus-micrometer-registry-prometheus"
```

   This command adds the following dependency to your **pom.xml**:

   pom.xml

```
<dependency>
    <groupId>io.quarkus</groupId>
  <artifactId>quarkus-micrometer-registry-prometheus</artifactId>
</dependency>
```

2. Enter the following command to display collected metrics on your terminal:

```
curl http://localhost:8080/q/metrics
```

3. (Optional) To enable the collection of metrics in JSON format using the Micrometer extension, add the following line to the **src/main/resources/application.properties** file:

```
quarkus.micrometer.export.json.enabled=true
```

4. Save the changes to the **application.properties** file.

5. Use the following command to view metrics in JSON format:

```
curl -i -H "Accept: application/json" -H "Content-Type: application/json"
http://localhost:8080/q/metrics
```

CHAPTER 3. CUSTOMIZED METRICS FOR YOUR QUARKUS APPLICATIONS

Micrometer provides an API that allows you to construct your own custom metrics. The most common types of meters supported by monitoring systems are gauges, counters, and summaries. The following sections build an example endpoint, and observes endpoint behavior using these basic meter types.

To register meters, you need a reference to a **MeterRegistry**, which is configured and maintained by the Micrometer extension. The **MeterRegistry** can be injected into your application as follows:

```
package org.acme.micrometer;

import io.micrometer.core.instrument.MeterRegistry;

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

@Path("/")
@Produces("text/plain")
public class ExampleResource {

    private final MeterRegistry registry;

    ExampleResource(MeterRegistry registry) {
        this.registry = registry;
    }
}
```

Micrometer does have conventions, such as meters must be created and named using dots to separate segments, for example, **a.name.like.this**. Micrometer then translates that name into the format that the selected registry prefers. Prometheus uses underscores, which means the previous name will appear as **a_name_like_this** in Prometheus-formatted metrics output.

Micrometer maintains an internal mapping between unique metric identifier and tag combinations and specific meter instances. Using **register**, **counter**, or other methods to increment counters or record values does not create a new instance of a meter unless that combination of identifier and tag or label value hasn't been seen before.

Gauges

Gauges measure a value that can increase or decrease over time, like the speedometer on a car. Gauges can be useful when monitoring the statistics for a cache or collection. Consider the following simple example that observes the size of a list:

```
    LinkedList<Long> list = new LinkedList<>();

    // Update the constructor to create the gauge
    ExampleResource(MeterRegistry registry) {
        this.registry = registry;
        registry.gaugeCollectionSize("example.list.size", Tags.empty(), list);
    }

    @GET
```

```
@Path("gauge/{number}")
public Long checkListSize(@PathParam("number") long number) {
    if (number == 2 || number % 2 == 0) {
        // add even numbers to the list
        list.add(number);
    } else {
        // remove items from the list for odd numbers
        try {
            number = list.removeFirst();
        } catch (NoSuchElementException nse) {
            number = 0;
        }
    }
    return number;
}
```

When using Prometheus, the value of the created gauge and the size of the list is observed when the Prometheus endpoint is visited. It is important to note that gauges are sampled rather than set; there is no record of how the value associated with a gauge might have changed between measurements.

Micrometer provides a few additional mechanisms for creating gauges. Note that Micrometer does not create strong references to the objects it observes by default. Depending on the registry, Micrometer either omits gauges that observe objects that have been garbage-collected entirely or uses **NaN** (not a number) as the observed value.

Never gauge something you can count. Gauges can be less straight-forward to use than counters. If what you are measuring can be counted (because the value always increments), use a counter instead.

Counters

Counters are used to measure values that only increase. In the example below, you will count the number of times you test a number to see if it is prime:

```
@GET
@Path("prime/{number}")
public String checkIfPrime(@PathParam("number") long number) {
    if (number < 1) {
        return "Only natural numbers can be prime numbers.";
    }
    if (number == 1 || number == 2 || number % 2 == 0) {
        return number + " is not prime.";
    }

    if ( testPrimeNumber(number) ) {
        return number + " is prime.";
    } else {
        return number + " is not prime.";
    }
}

protected boolean testPrimeNumber(long number) {
    // Count the number of times we test for a prime number
    registry.counter("example.prime.number").increment();
    for (int i = 3; i < Math.floor(Math.sqrt(number)) + 1; i = i + 2) {
        if (number % i == 0) {
            return false;
```

```
            }
        }
        return true;
    }
```

It might be tempting to add a label or tag to the counter indicating what value was checked. Remember that each unique combination of metric name (**testPrimeNumber**) and label value produces a unique time series. Using an unbounded set of data as label values can lead to a "cardinality explosion", an exponential increase in the creation of new time series.

It is possible to add a label that would convey a little more information, however. In the example below, adjustments were made and the counter was moved to add some labels.

```
@GET
@Path("prime/{number}")
public String checkIfPrime(@PathParam("number") long number) {
    if (number < 1) {
        registry.counter("example.prime.number", "type", "not-natural").increment();
        return "Only natural numbers can be prime numbers.";
    }
    if (number == 1 ) {
        registry.counter("example.prime.number", "type", "one").increment();
        return number + " is not prime.";
    }
    if (number == 2 || number % 2 == 0) {
        registry.counter("example.prime.number", "type", "even").increment();
        return number + " is not prime.";
    }

    if ( testPrimeNumber(number) ) {
        registry.counter("example.prime.number", "type", "prime").increment();
        return number + " is prime.";
    } else {
        registry.counter("example.prime.number", "type", "not-prime").increment();
        return number + " is not prime.";
    }
}

protected boolean testPrimeNumber(long number) {
    for (int i = 3; i < Math.floor(Math.sqrt(number)) + 1; i = i + 2) {
        if (number % i == 0) {
            return false;
        }
    }
    return true;
}
```

Looking at the data produced by this counter, you can tell how often a negative number was checked, or the number one, or an even number, and so on. Try the following sequence and look for **example_prime_number_total** in the plain text output. Note that the **_total** suffix is added when Micrometer applies Prometheus naming conventions to **example.prime.number**, the originally specified counter name.

```
# If you did not leave quarkus running in dev mode, start it again:
./mvnw compile quarkus:dev
```

```
curl http://localhost:8080/example/prime/-1
curl http://localhost:8080/example/prime/0
curl http://localhost:8080/example/prime/1
curl http://localhost:8080/example/prime/2
curl http://localhost:8080/example/prime/3
curl http://localhost:8080/example/prime/15
curl http://localhost:8080/q/metrics
```

Never count something you can time or summarize. Counters only record a count, which might be all that is needed. However, if you want to understand more about how a value is changing, a timer (when the base unit of measurement is time) or a distribution summary might be more appropriate.

Summaries and Timers

Timers and distribution summaries in Micrometer are very similar. Both allow you to record an observed value, which will be aggregated with other recorded values and stored as a sum. Micrometer also increments a counter to indicate the number of measurements that have been recorded and tracks the maximum observed value within a specified interval of time.

Distribution summaries are populated by calling the **record** method to record observed values, while timers provide additional capabilities specific to working with time and measuring durations. For example, we can use a timer to measure how long it takes to calculate prime numbers using one of the **record** methods that wraps the invocation of a Supplier function:

```
protected boolean testPrimeNumber(long number) {
    Timer timer = registry.timer("example.prime.number.test");
    return timer.record(() -> {
        for (int i = 3; i < Math.floor(Math.sqrt(number)) + 1; i = i + 2) {
            if (number % i == 0) {
                return false;
            }
        }
        return true;
    });
}
```

Micrometer will apply Prometheus conventions when emitting metrics for this timer. Prometheus measures time in seconds. Micrometer converts measured durations into seconds and includes the unit in the metric name, per convention. After visiting the prime endpoint a few more times, look in the plain text output for the following three entries: **example_prime_number_test_seconds_count**, **example_prime_number_test_seconds_sum**, and **example_prime_number_test_seconds_max**.

```
# If you did not leave quarkus running in dev mode, start it again:
./mvnw compile quarkus:dev

curl http://localhost:8080/example/prime/256
curl http://localhost:8080/q/metrics
curl http://localhost:8080/example/prime/7919
curl http://localhost:8080/q/metrics
```

Both timers and distribution summaries can be configured to emit additional statistics, like histogram data, precomputed percentiles, or service level objective (SLO) boundaries. Note that the count, sum, and histogram data can be re-aggregated across dimensions (or across a series of instances), while precomputed percentile values cannot.

CHAPTER 4. HTTP METRICS

The Micrometer extension automatically times HTTP server requests. Following Prometheus naming conventions for timers, look for **http_server_requests_seconds_count**, **http_server_requests_seconds_sum**, and **http_server_requests_seconds_max**. Dimensional labels have been added for the requested uri, the HTTP method (GET, POST, etc.), the status code (200, 302, 404, etc.), and a more general outcome field.

```
# HELP http_server_requests_seconds
# TYPE http_server_requests_seconds summary
http_server_requests_seconds_count{env="test",method="GET",outcome="SUCCESS",registry="prom
etheus",status="200",uri="/example/prime/{number}",} 6.0
http_server_requests_seconds_sum{env="test",method="GET",outcome="SUCCESS",registry="promet
heus",status="200",uri="/example/prime/{number}",} 0.007355093
http_server_requests_seconds_count{env="test",method="GET",outcome="SUCCESS",registry="prom
etheus",status="200",uri="/example/gauge/{number}",} 4.0
http_server_requests_seconds_sum{env="test",method="GET",outcome="SUCCESS",registry="promet
heus",status="200",uri="/example/gauge/{number}",} 0.005035393
# HELP http_server_requests_seconds_max
# TYPE http_server_requests_seconds_max gauge
http_server_requests_seconds_max{env="test",method="GET",outcome="SUCCESS",registry="promet
heus",status="200",uri="/example/prime/{number}",} 0.002110405
http_server_requests_seconds_max{env="test",method="GET",outcome="SUCCESS",registry="promet
heus",status="200",uri="/example/gauge/{number}",} 0.00239441
```

Ignoring endpoints

You can disable measurement of HTTP endpoints using the **quarkus.micrometer.binder.http-server.ignore-patterns** property. This property accepts a comma-separated list of simple regex match patterns identifying URI paths that should be ignored. For example, setting **quarkus.micrometer.binder.http-server.ignore-patterns=/example/prime/[0-9]+** will ignore a request to **http://localhost:8080/example/prime/7919**. A request to **http://localhost:8080/example/gauge/7919** would still be measured.

URI templates

The Micrometer extension will make a best effort at representing URIs containing path parameters in templated form. Using examples from above, a request to **http://localhost:8080/example/prime/7919** should appear as an attribute of **http_server_requests_seconds_*** metrics with a label of **uri=/example/prime/{number}**.

Use the **quarkus.micrometer.binder.http-server.match-patterns** property if the correct URL can not be determined. This property accepts a comma-separated list defining an association between a simple regex match pattern and a replacement string. For example, setting **quarkus.micrometer.binder.http-server.match-patterns=/example/prime/[0-9]+=/example/{jellybeans}** would use the value **/example/{jellybeans}** for the uri attribute any time the requested uri matches **/example/prime/[0-9]+**.

CHAPTER 5. RELIABILITY METRICS

Reliability metrics are used to express the reliability of a software product using quantitative measures. Which metric you use depends upon the type of system to which the reliability metrics applies and the requirements of the application domain. From a Site Reliability Engineering perspective, there are a few key metrics to focus on for Java applications.

Mean time to failure

Mean Time to Failure (MTTF) is the time interval between two successive failures. The time units you use to measure MTTF depends on the system and can also be defined by the number of transactions. For systems with large transactions, MTTF is typically consistent.

Mean Time to Repair

Mean Time to Repair (MTTR) is the average time it takes to track errors causing failure and repair them.

Mean Time Between Failure

When you combine MTTF and MTTR metrics, the result equals Mean Time Between Failure (MTBF). Time measurements are real-time and not the execution time that is included in MTTF.

Rate of Occurrence of Failure

The Rate of Occurrence of Failure (ROCOF) is the number of failures that occur in a unit time interval and focuses on the likelihood of frequently-occurring, unexpected events.

Probability of Failure on Demand

Probability of Failure on Demand (POFOD) is the probability that a system will fail when a service request is made. POFOD is an essential measure for safety critical systems and relevant for protection systems where services are demanded occasionally.

Availabiity

Availability measures the probability that the system is available for use at any given time. You must take into account the repair time and the restart time for the system.

CHAPTER 6. INTEGRATING WITH OPENSHIFT

You can enable your Quarkus application to use the Micrometer metrics library for runtime and application metrics. Micrometer acts like a facade between your applications and third parties like Prometheus, which is embedded in OpenShift. The integration of Quarkus with OpenShift enables you to expose not only embedded metrics that are automatically enabled, but also custom metrics that are registered as part of your application.

For more information on configuring a variety of metrics, see the Quarkus Micrometer Community Guide.

Prerequisites

- You have access to Getting started with the OpenShift CLI, 4.6 or later
- You have an OpenShift instance

Procedure

Complete the following instructions and use embedded Prometheus in OpenShift to expose metrics from your Quarkus applications:

1. Enabling monitoring for user-defined projects in OpenShift
2. Deploying your Quarkus application to OpenShift
3. Creating a service monitor in your OpenShift project

6.1. ENABLING MONITORING FOR USER-DEFINED PROJECTS IN OPENSHIFT

In Red Hat OpenShift Container Platform 4.6 or later, you can enable monitoring for your user-defined projects as well as default platform monitoring.

Prerequisites

- Have access to Getting started with the OpenShift CLI, 4.6 or later
- Have an OpenShift instance

Procedure

1. Go to Enabling monitoring for user-defined projects and follow specific instructions on how to enable user-defined project monitoring. In summary, you will create a ConfigMap in the namespace **openshift-monitoring**

cluster-monitoring-config.yaml:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cluster-monitoring-config
  namespace: openshift-monitoring
```

```
data:
  config.yaml: |
  enableUserWorkload: true
```

NOTE

If you are using OpenShift 4.5 or earlier, use:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cluster-monitoring-config
  namespace: openshift-monitoring
data:
  config.yaml: |
  techPreviewUserWorkload:
    enabled: true
```

After you complete the steps to enable monitoring for user-defined projects, OpenShift automatically creates a namespace, **openshift-user-workload-monitoring** that you will deploy when you begin Deploying your Quarkus application to OpenShift and Creating a service monitor in your OpenShift project.

6.2. DEPLOYING YOUR QUARKUS APPLICATION TO OPENSHIFT

After setting up the required infrastructure, you must enable Micrometer with Prometheus.

Prerequisites

- You have access to Getting started with the OpenShift CLI, 4.6 or later
- You have an OpenShift instance
- You have created a ConfigMap in the previous section, Enabling monitoring for user-defined projects in OpenShift

Procedure

1. Implement a REST API:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-resteasy</artifactId>
</dependency>
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-micrometer-registry-prometheus</artifactId>
</dependency>
```

2. Add the micrometer registry facade:

```
import javax.inject.Inject;
import javax.ws.rs.GET;
```

```
import javax.ws.rs.Path;

import io.micrometer.core.instrument.MeterRegistry;

@Path("/hello")
public class GreetingsResource {

    @Inject
    MeterRegistry registry;

    @GET
    public String sayHello() {
        registry.counter("greeting_counter").increment();

        return "Hello!";
    }
}
```

This Micrometer facade creates a counter that will be incremented every time you invoke the services. The registry helps to create custom metrics or use the metrics manually. You could also annotate methods as the following:

```
@GET
@Counted(value = "greeting_counter")
public String sayHello() {
    return "Hello!";
}
```

3. Run the application:

```
mvn compile quarkus:dev
```

4. Call your service:

```
curl http://localhost:8080/hello
```

The service should return with **Hello!**

5. Browse to **http://localhost:8080/q/metrics**. You should see the **greeting_counter** with count 1.0 (the one you just completed):

```
# HELP greeting_counter_total
#TYPE greeting_counter_total counter
greeting_counter_total 1.0
```

6. Deploy your Quarkus application into OpenShift by entering the extension **quarkus-openshift** into your **pom.xml**:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-openshift</artifactId>
</dependency>
```

7. Deploy your application into a newly created project called **my-project** in OpenShift:

```
oc new-project my-project

mvn clean package -Dquarkus.kubernetes.deploy=true -Dquarkus.openshift.expose=true -
Dquarkus.openshift.labels.app-with-metrics=quarkus-app
```

The **app-with-metrics** is explained in Creating a service monitor in your OpenShift project .

NOTE

If you are using an OpenShift with unsecure SSL, you also need to append **-Dquarkus.kubernetes-client.trust-certs=true** to the Maven command.

6.3. CREATING A SERVICE MONITOR IN YOUR OPENSHIFT PROJECT

Prometheus uses a pull model to get metrics from applications, which means it scrapes or watches endpoints to pull metrics. Although the previous procedure helped to expose your service in your OpenShift instance, you have not configured anything in Prometheus to scrape your service yet. This is why the service monitor is necessary.

A service monitor is a custom resource that you must create in the same project or namespace where your service is running: **my-project**.

Procedure

1. Set up your **service-monitor.yaml**:

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
    k8s-app: prometheus-app-monitor
  name: prometheus-app-monitor
  namespace: my-project
spec:
  endpoints:
  - interval: 30s
    targetPort: 8080
    scheme: http
  selector:
    matchLabels:
      app-with-metrics: 'quarkus-app'
```

2. Apply your service-monitor.yaml:

```
oc apply -f service-monitor.yaml
```

This command creates a service monitor named **prometheus-app-monitor** that will select applications with the label **app-with-metrics: quarkus-app**. This label was added during the Deploying your Quarkus application to OpenShift procedure. OpenShift calls the endpoint **/metrics** for all the services labeled with **app-with-metrics: quarkus-app**.

3. To use your service monitor:

a. Call your greetings service: **curl http://quarkus-micrometer-my-project.ocp.host/hello**. This increments your **greeting_counter_total** counter.

b. To see the metrics, browse to the OpenShift Console and select the **Developer > Monitoring** view.

c. Select the **Metrics** tab.

d. In the **Custom Query** field, enter **greeting_counter_total**.

The metrics display in the table below the Custom Query field.

CHAPTER 7. ADDITIONAL RESOURCES

- Micrometer Application Monitoring
- Micrometer Configuration Reference
- Open Shift Platform 4.6: Enabling monitoring for user-defined projects
- Developing and compiling your Quarkus applications with Apache Maven
- Configuring your Quarkus applications
- Deploying your Quarkus applications to OpenShift
- Testing your Quarkus applications
- Apache Maven Project
- JUnit 5
- REST-assured

Revised on 2021-08-23 08:35:54 UTC