# Metrics Component

## Metrics Component

**Available as of Camel 2.14**

The **`metrics:`** component allows to collect various metrics directly from Camel routes. Supported metric types are counter, histogram, meter and timer. Metrics provides simple way to measure behavior of application. Configurable reporting backend is enabling different integration options for collecting and visualizing statistics.

The **`metrics`** component also provides a **`MetricsRoutePolicyFactory`** that allows route statistics to be exposed using Codahale metrics. See bottom of page for details.

Maven users will need to add the following dependency to their **`pom.xml`** for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-metrics</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

### URI Format

```
metrics:[ meter | counter | histogram | timer ]:metricname[?options]
```

### Metric Registry

Camel Metrics Component uses by default **`MetricRegistry`** with **`Slf4jReporter`** and 60 second reporting interval. Default registry can be replaced with custom one by providing bean with name **`metricRegistry`** in Camel registry.

For example using Spring Java Configuration:

```
@Configuration
public static class MyConfig extends SingleRouteCamelConfiguration {

    @Bean
    @Override
    public RouteBuilder route() {
        return new RouteBuilder() {
            @Override
            public void configure() throws Exception {
                // define Camel routes here
            }
        };
    }

    @Bean(name = MetricsComponent.METRIC_REGISTRY_NAME)
    public MetricRegistry getMetricRegistry() {
        MetricRegistry registry = ...;
        return registry;
    }
}
```

**`MetricRegistry`** uses internal thread(s) for reporting. There is no public API in DropWizard version 3.0.1 for users to clean up on exit. Thus using Camel Metrics Component leads to Java classloader leak and my cause **`OutOfMemoryErrors`** in some cases.

### Usage

Each metric has type and name. Supported types are counter, histogram, meter and timer. Metric name is simple string. If metric type is not provided then type meter is used by default.

#### Headers

Metric name defined in URI can be overridden by using header with name **`CamelMetricsName.`**

For example

```
from("direct:in")
    .setHeader(MetricsConstants.HEADER_METRIC_NAME, constant("new.name"))
    .to("metrics:counter:name.not.used")
    .to("direct:out");
```

will update counter with name **`new.name`** instead of **`name.not.used`**.

All **`metrics`** specific headers are removed from the message once Metrics endpoint finishes processing of exchange. While processing the exchange **`metrics`** endpoint any exceptions thrown are caught and written to as a log entry at level **`warn`**.

## Metrics Type Counter

```
metrics:counter:metricname[?options]
```

### Options

| Name | Default | Description |
|---|---|---|
| `increment` | - | Long value to add to the counter |
| `decrement` | - | Long value to subtract from the counter |

If neither **`increment`** or **`decrement`** is defined then counter value will be incremented by one. If **`increment`** and **`decrement`** are both defined only **`increment`** operation is called.

```
// update counter simple.counter by 7
from("direct:in")
    .to("metric:counter:simple.counter?increment=7")
    .to("direct:out");
```

```
// increment counter simple.counter by 1
from("direct:in")
    .to("metric:counter:simple.counter")
    .to("direct:out");
```

```
// decrement counter simple.counter by 3
from("direct:in")
    .to("metric:counter:simple.counter?decrement=3")
    .to("direct:out");
```

### Headers

Message headers can be used to override **`increment`** and **`decrement`** values specified in the **`metrics`** component URI.

| Name | Description | Expected type |
|---|---|---|
| `CamelMetricsCounterIncrement` | Override increment value in URI | `Long` |
| `CamelMetricsCounterDecrement` | Override decrement value in URI | `Long` |

```
// update counter simple.counter by 417
from("direct:in")
    .setHeader(MetricsConstants.HEADER_COUNTER_INCREMENT, constant(417L))
    .to("metric:counter:simple.counter?increment=7")
    .to("direct:out");
```

```
// updates counter using simple language to evaluate body.length
from("direct:in")
    .setHeader(MetricsConstants.HEADER_COUNTER_INCREMENT, simple("${body.length}"))
    .to("metrics:counter:body.length")
    .to("mock:out");
```

## Metric type histogram

```
metrics:histogram:metricname[?options]
```

### Options

| Name | Default | Description |
|---|---|---|
| `value` | - | Value to use in histogram |

If no **`value`** is not set nothing is added to histogram and warning is logged.

```
// adds value 9923 to simple.histogram
from("direct:in")
    .to("metric:histogram:simple.histogram?value=9923")
    .to("direct:out");
```

```
// nothing is added to simple.histogram; warning is logged
from("direct:in")
    .to("metric:histogram:simple.histogram")
    .to("direct:out");
```

### Headers

Message header can be used to override value specified in **`metrics`** component URI.

| Name | Description | Expected type |
|---|---|---|
| `CamelMetricsHistogramValue` | Override histogram value in URI | `Long` |

```
// Adds value 992 to simple.histogram
from("direct:in")
    .setHeader(MetricsConstants.HEADER_HISTOGRAM_VALUE, constant(992L))
    .to("metric:histogram:simple.histogram?value=700")
    .to("direct:out")
```

## Metric type meter

```
metrics:meter:metricname[?options]
```

### Options

| Name | Default | Description |
|---|---|---|
| `mark` | - | Long value to use as mark |

If **`mark`** is not set then **`meter.mark()`** is called without argument.

```
// marks simple.meter without value
from("direct:in")
    .to("metric:simple.meter")
    .to("direct:out");
```

```
// marks simple.meter with value 81
from("direct:in")
    .to("metric:meter:simple.meter?mark=81")
    .to("direct:out");
```

### Headers

Message header can be used to override **`mark`** value specified in **`metrics`** component URI.

| Name | Description | Expected type |
|---|---|---|
| `CamelMetricsMeterMark` | Override mark value in URI | `Long` |

```
// updates meter simple.meter with value 345
from("direct:in")
    .setHeader(MetricsConstants.HEADER_METER_MARK, constant(345L))
    .to("metric:meter:simple.meter?mark=123")
    .to("direct:out");
```

## Metrics type timer

```
metrics:timer:metricname[?options]
```

### Options

| Name | Default | Description |
|---|---|---|
| `action` | - | start or stop |

If no **`action`** or invalid value is provided then warning is logged without any timer update. If action **`start`** is called on already running timer or **`stop`** is called on not running timer then nothing is updated and warning is logged.

```
// measure time taken by route "calculate"
from("direct:in")
    .to("metrics:timer:simple.timer?action=start")
    .to("direct:calculate")
    .to("metrics:timer:simple.timer?action=stop");
```

**`TimerContext`** objects are stored as **`Exchange`** properties between different **`metrics`** component calls.

### Headers

Message header can be used to override action value specified in **`metrics`** component URI.

| Name | Description | Expected type |
|---|---|---|
| `CamelMetricsTimerAction` | Override timer action in URI | **`org.apache.camel.component.metrics.timer.TimerEndpoint.TimerAction`** |

```
// sets timer action using header
from("direct:in")
    .setHeader(MetricsConstants.HEADER_TIMER_ACTION, TimerAction.start)
    .to("metric:timer:simple.timer")
    .to("direct:out");
```

## MetricsRoutePolicyFactory

This factory allows to add a RoutePolicy for each route which exposes route utilization statistics using CodaHale metrics. This factory can be used in Java and XML as the examples below demonstrates.

Instead of using the **MetricsRoutePolicyFactory** you can define a **MetricsRoutePolicy** per route you want to instrument, in case you only want to instrument a few selected routes.

From Java you just add the factory to the **CamelContext** as shown below:

```
context.addRoutePolicyFactory(new MetricsRoutePolicyFactory());
```

And from XML DSL you define a **<bean>** as follows:

```
  <!-- use camel-metrics route policy to gather metrics for all routes -->
  <bean id="metricsRoutePolicyFactory" class="org.apache.camel.component.metrics.routepolicy.
MetricsRoutePolicyFactory"/>
```

The **MetricsRoutePolicyFactory** and **MetricsRoutePolicy** supports the following options:

| Name | Default | Description |
|---|---|---|
| useJmx | false | Whether to report fine grained statistics to JMX by using the **com.codahale.metrics.JmxReporter**.<br><br>Notice that if JMX is enabled on CamelContext then a **MetricsRegistryService** mbean is enlisted under the services type in the JMX tree. That MBean has a single operation to output the statistics using JSON. Setting **useJmx** to true is only needed if you want fine grained MBeans per statistics type. |
| jmxDomain | org.apache.camel.metrics | The JMX domain name. |
| prettyPrint | false | Whether to use pretty print when outputting statistics in JSON format. |
| metricsRegistry | | Allow to use a shared **com.codahale.metrics.MetricRegistry**. If one is not configured Camel will create a shared instance for use by the **CamelContext**. |
| rateUnit | TimeUnit.SECONDS | The unit to use for rate in the metrics reporter or when dumping the statistics as JSON. |
| durationUnit | TimeUnit.MILLISECONDS | The unit to use for duration in the metrics reporter or when dumping the statistics as JSON. |
| namePattern | ##name##.##routeId##.##type## | **Camel 2.17**: The name pattern to use. Uses dot as separators, but you can change that. The values **##name##**, **##routeId##**, and **##type##** will be replaced with actual value. Where **###name###** is the name of the **CamelContext.** **###routeId###** is the id of the route, where **###type###** is the value of responses. |

In Java you can get the **com.codahale.metrics.MetricRegistry** from the **org.apache.camel.component.metrics.routepolicy.MetricsRegistryService** as shown below:

```
MetricRegistryService registryService = context.hasService(MetricsRegistryService.class);

if (registryService != null) {
  MetricsRegistry registry = registryService.getMetricsRegistry();
  // ...
}
```

## MetricsMessageHistoryFactory

**Available as of Camel 2.17**

This factory allows to use metrics to capture Message History performance statistics while routing messages. It works by using a metrics Timer for each node in all the routes. This factory can be used in Java and XML as the examples below demonstrates.

In Java set the factory on the **CamelContext** as shown below:

```
context.setMessageHistoryFactory(new MetricsMessageHistoryFactory());
```

And from XML DSL you define a **<bean>** as follows:

```
  <!-- use camel-metrics message history to gather metrics for all messages being routed -->
  <bean id="metricsMessageHistoryFactory" class="org.apache.camel.component.metrics.messagehistory.
MetricsMessageHistoryFactory"/>
```

The following options is supported on the factory:

| Name | Default | Description |
|---|---|---|
| useJmx | false | Whether to report fine grained statistics to JMX by using the **com.codahale.metrics.JmxReporter**.<br><br>Notice that if JMX is enabled on CamelContext then a **MetricsRegistryService** MBean is enlisted under the services type in the JMX tree. That MBean has a single operation to output the statistics using JSON. Setting **useJmx** to true is only needed if you want fine grained MBeans per statistics type. |
| jmxDomain | org.apache.camel.metrics | The JMX domain name. |
| prettyPrint | false | Whether to use pretty print when outputting statistics in JSON format. |
| metricsRegistry | | Allow to use a shared **com.codahale.metrics.MetricRegistry**. If one is not provided Camel will create a shared instance for use by the **CamelContext**. |
| rateUnit | TimeUnit.SECONDS | The unit to use for rate in the metrics reporter or when dumping the statistics as JSON. |
| durationUnit | TimeUnit.MILLISECONDS | The unit to use for duration in the metrics reporter or when dumping the statistics as JSON |
| namePattern | ##name##.##routeId##.###id###.##type## | The name pattern to use. Uses dot as separators, but you can change that. The values **##name##**, **##routeId##**, **##type##**, and **###id###** will be replaced with actual value. Where **###name###** is the name of the **CamelContext.** **###routeId###** is the name of the route. The **###id###** pattern represents the node id, where **###type###** is the value of history. |

At runtime the metrics can be accessed from Java API or JMX which allows to gather the data as JSON output.

In Java, get the service from the **CamelContext** as shown:

```
MetricsMessageHistoryService service = context.hasService(MetricsMessageHistoryService.class);
String json = service.dumpStatisticsAsJson();
```

And the JMX API the MBean is registered in the **type=services tree with name=MetricsMessageHistoryService**.