Java: Adding custom metrics to Spring Boot Micrometer Prometheus endpoint

June 29, 2022
Categories: Development, Java

If you have enabled Actuator and the 'micrometer-registry-prometheus' dependency in your Spring Boot application, then you will have a new '/actuator/prometheus' web endpoint that returns general information about threads, garbage collection, disk, and memory.

This information is delivered in standard prometheus formatting as plaintext, with one metric per line.

This is exactly the type of information you want to provide to a Prometheus scrape job, but in addition to these generic metrics, you will likely also want to return service specific metrics at this same endpoint.

This can be done by using constructor injection of the MeterRegistry into a custom class, and then using this object to create a set of custom Gauge/Counter that will return runtime values.

Spring Boot prerequisites

If you are reading this article, I'm assuming you have enabled the Actuator package already, and that you have already proven that you can can reach endpoints such as /actuator and /actuator/health for your Spring Boot web application.

If not, go through this article from Baeldung for the basic setup.

To enable the '/actuator/prometheus' endpoint, add the micrometer Prometheus dependency in build.gradle

```
dependencies {
   implementation 'org.springframework.boot:spring-boot-starter-
actuator'
   runtimeOnly 'io.micrometer:micrometer-registry-
prometheus:1.9.0'
   ...
}
```

Or pom.xml if using Maven.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
  <scope>compile</scope>
</dependency>
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-registry-prometheus</artifactId>
  <scope>runtime</scope>
</dependency>
```

And expose the management endpoints in application.properties

```
management.endpoints.enabled-by-default=true
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details: always
```

These changes above, without any custom steps, will expose an endpoint at '/actuator/prometheus' that will provide an entire set of generic metrics for JVM threads, garbage collection metrics, cpu, and disk usage. Do not move forward with the customization until you prove this successfully.

Add custom metric from web Controller

If the metric you want to expose is related to the web controller level, then create a constructor for the @Controller that injects the MeterRegistry.

```
import io.micrometer.core.instrument.MeterRegistry;
import io.micrometer.core.instrument.Gauge;

@RestController
@RequestMapping("/api/user")
public class UserController {

  // list of our users for this simple example
  private List<User> userListV1;

  // supplies user count
  public Supplier<Number> fetchUserCount() {
    return ()->userListV1.size();
  }

  // constructor injector for exposing metrics at Actuator
/prometheus
  public UserController(MeterRegistry registry) {

    Gauge.builder("usercontroller.usercount",fetchUserCount()).
      tag("version","v1").
      description("usercontroller descrip").
      register(registry);
  }

...
```

In this particular example, we are using a @RestController. The constructor is where we accept the MeterRegistry that allows us to add Gauge or Counter that will be exposed at the '/actuator/prometheus' endpoint.

The constructor is only called once, so instead of providing a value for our metric directly in the Gauge.builder, we must instead provide a reference to a Supplier<Number> that is dynamically invoked when the Gauge is queried.

Adding a tag to this Gauge adds a dimension to the metric so it looks like below when pulling up '/actuator/prometheus'.

```
# HELP usercontroller_usercount usercontroller descrip
# TYPE usercontroller_usercount gauge
usercontroller_usercount{version="v1",} 3.0
```

The value of this metric will change as the size returned by 'fetchUserCount()' changes.

Here is the full UserController.java source code.

Add timed metric from web Controller

Beyond Gauge/Counter, another type of metric that can be returned is for timing. We can annotate a Controller method with @Timed to get statistics on its response time and quantile response times.

For example, here is a Controller that records metrics on the GET method for '/api/user' that retrieves a list of all users. It will record metrics for the 50% and 90% quartile((e.g. max millisecond response for 50% of requests, max millisecond response for 90% of requests).

```
import io.micrometer.core.annotation.Timed;

@RestController
@RequestMapping("/api/user")
public class UserController {

  @Timed(value="user.get.time",description="time to retrieve
users",percentiles={0.5,0.9})
  @GetMapping
  public Iterable<User> findAllUsers() {
  ...
  }
```

It requires at least one invocation of '/api/user' in order to populate this metrics, but once that is made, you should see a set of multi-dimensional metrics like below at '/actuator/prometheus'.

```
# HELP user_get_time_seconds_max time to retrieve users
# TYPE user_get_time_seconds_max gauge
user_get_time_seconds_max{class="org.fabianlee.springmicrowithact
uator.user.UserController",exception="none",method="findAllUsers"
,} 0.023658932
# HELP user_get_time_seconds time to retrieve users
# TYPE user_get_time_seconds summary
user_get_time_seconds{class="org.fabianlee.springmicrowithactuato
r.user.UserController",exception="none",method="findAllUsers",qua
ntile="0.5",} 0.023068672
user_get_time_seconds{class="org.fabianlee.springmicrowithactuato
r.user.UserController",exception="none",method="findAllUsers",qua
ntile="0.9",} 0.023068672
user_get_time_seconds_count{class="org.fabianlee.springmicrowitha
ctuator.user.UserController",exception="none",method="findAllUser
s",} 1.0
user_get_time_seconds_sum{class="org.fabianlee.springmicrowithact
uator.user.UserController",exception="none",method="findAllUsers"
,} 0.023658932
```

If you have problems with @Timed metrics not showing up, try adding a Configuration object that returns a TimedAspect like this. Or you can add it to the SpringBootApplication class like this.

Add metric from custom class

If you are exposing service level or aggregate metrics, or simply believe that instrumenting your Controller is mixing concerns that should be separated, then you can instead use an independent custom class to expose metrics.

Once again, you will need to inject the MeterRegistry in the constructor and add any custom Counter or Gauge you want exposed.

```
import org.springframework.stereotype.Component;
import io.micrometer.core.instrument.MeterRegistry;

@Component
public class MyMetricsCustomBean {

  public MyMetricsCustomBean(MeterRegistry registry) {
    // custom Gauge/Counter added here
  }
...
```

You can @Autowired any object in the Spring registry that you need to populate the custom metrics. For example, if you wanted to expose the UserController user count then you could Autowire it like below and add a Gauge.

```
import org.springframework.stereotype.Component;
import io.micrometer.core.instrument.MeterRegistry;

@Component
public class MyMetricsCustomBean {

  @Lazy
  @Autowired
  protected UserController userController;

  public Supplier<Number> fetchUserCount() {
    return ()->userController.fetchUserCount();
  }
  public MyMetricsCustomBean(MeterRegistry registry) {
    // simple, non-dimensional prometheus metric

Gauge.builder("number.of.users",fetchUserCount()).register(regist
ry);
  }
...
```

This will add the custom Gauge metric at '/actuator/prometheus' like below.

```
number_of_users 3.0
```

MultiGauge for dimensional metrics

For non-dimensional metrics, we can provide a function that implements Supplier<Number> to the Gauge as above, but for dimensional metrics we must take a different approach. The MultiGauge allows us to define the dimensions in the constructor and keep a reference that can then be populated with a Iterable set of values.

Assuming a ProductRepository that has methods for accessing the database and can return a Collection of Product objects that have low inventory counts, here is how a dimensional Gauge can be populated.

```
import org.springframework.stereotype.Component;
import io.micrometer.core.instrument.MeterRegistry;

@Component
public class MyMetricsCustomBean {

  // multigauge for low inventory (dimensions on product id and
name)
  MultiGauge lowInventoryCounts = null;
  // access to Database for metrics
  @Lazy
  @Autowired
  protected ProductRepository productRepository;

  // MultiGauge repopulated every time this is invoked
  public void updateLowInventoryGauges() {
      boolean overWrite = true;

      // create MultiGauge.Row for each product with low
inventory count
      lowInventoryCounts.register(

productRepository.findProductWithLowInventoryCount().stream().
          map(
              (Product p) ->
MultiGauge.Row.of(Tags.of("pid",""+p.getId(),"pname",p.getName())
,p.getCount())
          ).
          collect(Collectors.toList()
          )
      ,overWrite);

  }

  public MyMetricsCustomBean(MeterRegistry registry) {
    // definition of MultiGauge only, no values provided
    lowInventoryCounts =
MultiGauge.builder("low.inventory.count").tag("pid","pname").regi
ster(registry);
  }

...
```

This constructs the MultiGauge in the constructor and holds a reference to it, so that every time the updateLowInventoryGauges() method is invoked, it can populate the actual dimension tags and values.

The only hitch is that updateLowInventoryGauges() must be invoked for these metrics to be generated. This could be done any way you choose, for example with a Spring timer, or upon receiving an event or message, or for this example being invoked from the ProductController when a product is updated or sold.

Below is an example output from '/actuator/prometheus' when enough Coffee Cups have been sold (<3 in stock). You can see that the database product ID and name are dimensions, and there is only 1 cup left on the shelf.

```
# HELP low_inventory_count
# TYPE low_inventory_count gauge
low_inventory_count{pid="2",pname="Coffee Cup",} 1.0
```

Here is the full source for MyMetricsCustomBean.java where MultiGauge metrics are exposed via an independent custom class.

REFERENCES

Spring docs, Actuator

Baeldung, Actuator 2.0

micrometer docs, concepts and code examples

tutorialworks, Spring Boot and Prometheus Micrometer

tutorialworks, github project with example adding TimedAspect in SpringBootApplication

frankel.ch, Spring Boot Counter and Gauge explanation

Prometheus docs, metric output syntax