Application Metrics With Spring Boot Actuator

3 years ago 10/01/2015 4 16

Spring Boot, Spring Actuator, AOP, Java

Update 12/2017: It will need an update/rewrite since Spring Boot 2.0 is coming.

Having metrics collected is vital for ...just anything, besides relationships maybe :) My favourite quote by Deming goes like this: *"You can't manage what you can't measure"*. Without it either your experience, prediction and planning skills are so awesome that everything works as expected, or you're just deluding yourself. It's hard data that gives you feedback to confront your actions with reality. It's not only important on company-wide level, or in project management, processes, or when counting conversions in Google Analytics. There are metrics you can collect down on the application level, so can have insight on how it is performing, being used and that it works at all. A nice tool exists for Spring Boot apps to do that, and this is Spring Actuator I'm going to write about today.

The example of collecting and reporting metrics is as usual on GitHub.

Dependencies

Besides standard Spring Boot dependencies a starter for Actuator should be present in `pom.xml`:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Doing the above exposes some interesting HTTP endpoints for the application that can be monitored or collected. The full list is in Spring Boot reference documentation, but the ones I'm going to talk about are:

- `/health` - gives very reassuring `{"status":"UP"}` and it's used for health checks
- `/metrics` - it's used to read metrics collected by application, and by default returns a list of "system" metrics, for example:

  { "mem":144896, "mem.free":58557, "processors":4, "uptime":215637, "instance.uptime":208790, "systemload.average":1.91015625, "heap.committed":144896, "heap.init":131072, "heap.used":86338, "heap":1864192, "threads.peak":28, "threads.daemon":24, "threads":28, "classes":8552, "classes.loaded":8552, "classes.unloaded":0, "gc.ps_scavenge.count":25, "gc.ps_scavenge.time":134, "gc.ps_marksweep.count":3, "gc.ps_marksweep.time":431, "httpsessions.max":-1, "httpsessions.active":0 }

Customizing endpoints

You can change how those endpoints are exposed using `application.properties`, the most common settings:

- `management.port=8081` - you can expose those endpoints on port other than the one application is using (8081 here).
- `management.address=127.0.0.1` - you can only allow to access by IP address (localhost here).
- `management.context-path=/actuator` - allows you to have those endpoints grouped under specified context path rather than root, i.e. `/actuator/health`.
- `endpoints.health.enabled=false` - allows to enable/disable specified endpoint by name, here `/health` is disabled.

The information exposed by endpoints is most of the time sensitive. While `/health` is usually harmless to be exposed, `/metrics` would be too much. Fortunately, you can use Spring Security for that purpose. If it's present on the classpath, it is automatically picked up and used for Actuator. It involves adding a dependency to the `pom.xml`:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

After that, by default, you have basic http security enabled all over your application, allowing access only to the user named `user` and a password that pops up when the application starts:

```
Using default security password: ***PASSWORD***
```

This does the job, but rarely it is what you want, as your application might not need security besides that at all. Fortunately you can disable basic security it in `application.properties`, so that it leaves only the sensitive Actuator endpoints secured and leaves the rest open for access:

```
security.basic.enabled=false
```

You can also set up a new username, or a password if you don't want it to be different on each start:

```
security.user.name=admin
security.user.password=new_password
```

In case you're using the security features across the application and decided to secure those endpoints yourself, you can disable default security for Actuator:

Or just force it to allow access for the users authenticated by the application and having authority a.k.a. role.

```
management.security.role=ADMIN
```

As a bottom line it's suffice to say that the marriage between Actuator and Security is useful and flexible enough to customize it as you want.

Custom health checks

The basic idea for health checks is that they can provide more insightful information to you on the application's health. Besides checking if the application is UP or DOWN, which is done by default, you can add checks for things like database connectivity or whatever suits you. This is in fact what is being done when you add other Spring Boot starters, as they often provide additional health checks.

To create your your own health check, just do as stated in the reference documentation:

```
@Component
public class MyHealth implements HealthIndicator {
    @Override
    public Health health() {
        int errorCode = check(); // perform some specific health check
        if (errorCode != 0) {
            return Health.down().withDetail("Error Code", errorCode).build();
        }
        return Health.up().build();
    }
}
```

on the `/health` endpoint, so the application can be monitored for them.

Custom metrics

Similar to health checks, there is a set of metrics available to you already, that can be extended by other Spring Boot starters being used. For example Spring Boot MVC provides metrics for number of calls to each exposed HTTP method and their execution time. You can also add your metrics yourself, for example deeper, in service layer of your application.

In the example application I have a `GreetingServiceImpl` with a method that returns one of the greetings based on its parameter or throwing exception if someone requests a greeting that's not there:

```
@Service
class GreetingServiceImpl implements GreetingService {


    private static final String[] GREETINGS = {
            "Yo!", "Hello", "Good day", "Hi", "Hey"
    };


    @Override
    public String getGreeting(int number) {
        if (number < 1 || number > GREETINGS.length) {
            throw new NoSuchElementException(String.format("No greeting #%d", number));
        }
        return GREETINGS[number - 1];
    }


}
```

requesting each greeting and how many times the exception was thrown. To do so, Actuator provides `CounterService` with a simple interface that can be used to create and increase counters. The most basic usage would be:

```
@Service
class GreetingServiceImpl implements GreetingService {

    private final CounterService counterService;

    @Autowired
    public GreetingServiceImpl(CounterService counterService) {
        this.counterService = counterService;
    }

    private static final String[] GREETINGS = {
            "Yo!", "Hello", "Good day", "Hi", "Hey"
    };

    @Override
    public String getGreeting(int number) {
        if (number < 1 || number > GREETINGS.length) {
            counterService.increment("counter.errors.get_greeting");
            throw new NoSuchElementException(String.format("No greeting #%d", number));
        }
        counterService.increment("counter.calls.get_greeting");
        counterService.increment("counter.calls.get_greeting." + (number - 1));
        return GREETINGS[number - 1];
    }
```

After calling the method the `counter.errors.*` and `counter.calls.*` will appear on `/metrics` so you can have your valuable information.

Besides `CounterService` the other one provided by default is `GaugeService` that is used to collect a single `double` value, i.e. a measured execution time. You can also create and use your own implementations of these two.

Collecting metrics in Aspects

Handling counters by services like above can be nasty as it pollutes the code with things that lies apart from its main concern. For things like that aspect-oriented programming was invented. It allows you to separate handling metrics by a separate service intercepting the calls to measured methods.

To use AOP in Spring Boot application this needs to be added to `pom.xml`:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```


And the aspect to measure usage of `GreetingService.getGreeting()` would be written like that:

```
@Aspect
@Component
class GreetingServiceMetricsAspect {

    private final CounterService counterService;
```

```
        this.counterService = counterService;
    }

    @AfterReturning(pointcut = "execution(* eu.kielczewski.example.service.greeting.Gree
    public void afterCallingGetGreeting(int number) {
        counterService.increment("counter.calls.get_greeting");
        counterService.increment("counter.calls.get_greeting." + number);
    }

    @AfterThrowing(pointcut = "execution(* eu.kielczewski.example.service.greeting.Greet
    public void afterGetGreetingThrowsException(NoSuchElementException e) {
        counterService.increment("counter.errors.get_greeting");
    }

}
```

An aspect is a `@Component` also annotated by `@Aspect`. It has two methods annotated by:

- `@AfterReturning` - it is executed after the method returns a value and no exception is thrown. The parameter is extracted from the method call.
- `@AfterThrowing` - it is executed after the method throws the exception.

There is more to Aspect-Oriented Programming that this basic example shows, but this is just to show you that it can be done like that. This is a powerful tool in general.

For example you could make this more generic then shown, for example to count a number of calls you could create a custom annotation which you'd annotate your methods with. Then in the aspect, you can intercept the calls to those annotated methods increasing a counter with a name derived from method name.

Besides having them available through the HTTP endpoint you can also actively push the metrics away for other tools to be collected and stored for further analysis. The access to currently collected metrics is provided through `MetricsRepository`.

One can ask what's the hassle in using `CounterService` and `MetricRepository` if the metrics are to be eventually exported and processed by other tools. This could be done the moment when the method executes. The answer to this is that although it would work, the exporting operation is slow, so you don't want to do it during a method call. It's better to have separate task to export them, that can be triggered by scheduler, or whatever you wish.

A crude example is given in the example application, when the metrics are dumped to the JSON-enabled logger. This can be later collected by Logstash and pushed to ElasticSearch to be analyzed. The code to do that is like that:

```
@Service
class MetricExporterService {
    private static final Logger LOGGER = LoggerFactory.getLogger(MetricExporterService.c
    private final MetricRepository repository;

    @Autowired
    public MetricExporterService(MetricRepository repository) {
        this.repository = repository;
    }

    @Scheduled(initialDelay = 60000, fixedDelay = 60000)
    void exportMetrics() {
        repository.findAll().forEach(this::log);
    }
```

```
            repository.reset(m.getName());
        }
    }
```

What it does is that a method, scheduled to be executed every minute, reads everything from `MetricRepository` and dumps each metric to the logger in a separate JSON field `metric`. After that the metric is reset. In the logs it looks like this:

```
{
    "@timestamp":"2015-01-10T14:01:16.551+00:00",
    "@version":1,
    "message":"Reporting metric counter.calls.get_greeting=1",
    "logger_name":"eu.kielczewski.example.service.metric.MetricExporterService",
    "thread_name":"pool-1-thread-1",
    "level":"INFO",
    "level_value":20000,
    "HOSTNAME":"localhost",
    "metric": {
        "name": "counter.calls.get_greeting",
        "value":1,
        "timestamp": 1420898429647
    }
}
```

To make `@Scheduled` annotation to work you have to put `@EnableScheduling` somewhere in the configuration, like in `Application.java` file in the example.

To see how to enable logging to JSON, please take a look on this article about logging.

Closing remarks

classpath the metrics are made available through a `MetricRegistry` exposed as a Spring Bean. This not only gives you access to more metric types like histograms, but also you can use it to export metrics to tools like Graphite.
- You can use a Java-Zabbix bridge or agent implementation and push them to Zabbix
- It also integrates with JMX messaging, so they can be pushed out to the message broker.

Spring Boot Actuator is also an audit framework. The same things that are done for metrics can also be done for audit messages. The audit events are treated as other Spring application events just being an instances of `AuditApplicationEvent`, so you can push them to `ApplicationEventPublisher`. Then they can be read using `AuditEventRepository`.

All of this provides nice and ready to use library that enables you to have more insight on the application in the runtime, that you can use for both monitoring purposes and gathering 'business intelligence' as well.

Polite Notice - if you have a questions concerning implementation details in your own projects then you're much better off asking them on Stack Overflow. More people to help you this way.