Métricas y micrometer

Características de un sistema de monitoreo

Existe 3 características de un sistema de monitoreo que es importante que se entiendan:

- **Dimensionality** (Dimensionamiento) : Si el sistema de metricas soporta ser enriquecido con tags (llave / valor). Si el sistema no es dimensional, es jerárquico, lo que significa que no solo soporta nombres planos de nombres de métricas. Micrometer aplana el conjunto de tags y las agrega al nombre.
- **Rate aggregation** (Agrupado de medidas): En este contexto, la agregación o resumen dentro de un periodo de tiempo.
- **Publishing** (Publicación): Algunos sistemas esperan consumir métricas desde las aplicaciones y otros esperan que se publiquen en intervalos regulares.

Existen otras variaciones en las expectativas de los sistemas de monitoreo, para el caso de estudio de este curso se utilizará **Prometheus**.

Registry

Una **Medida** es la interfaz para recolectar un conjunto de datos sobre tu aplicación. En Micrometer una medida se crea y mantiene en un **MeterRegistry**. Cada sistema de monitoreo tiene su propia implementación del **MeterRegistry**.

La implementación de Micrometer del **MeterRegistry** se llama **SimpleMeterRegistry,** la cual mantiene el último valor de cada medida en memoria y no exporta la información a ningún lugar. Ideal en caso de que no tengas alguna preferencia con un sistema de monitoreo:

```
MeterRegistry registry=new SimpleMeterRegistry();
```

En aplicaciones basadas en **Spring SimpleMeterRegistry** es inyectado por default, si se incluye la dependencia de Prometheus el registry por default es **PrometheusMeterRegistry**.

Composite registries

Micrometer provee un **CompositeMeterRegistry** donde puedes agregar múltiples registros, permitiendo publicar métricas a más de un sistema de monitoreo de forma simultanea.

```
CompositeMeterRegistry composite = new
CompositeMeterRegistry();

Counter compositeCounter =
composite.counter("devs4j.students");

compositeCounter.increment();

SimpleMeterRegistry simple = new SimpleMeterRegistry();
composite.add(simple);

compositeCounter.increment();

log.info("Devs4j students {}",compositeCounter.count());
```

Incrementos son operaciones que no hacen nada hasta que hay un **MeterRegistry** en el **CompositeMeterRegistry**, por lo que la salida anterior imprimirá 1.0.

Global registry

Micrometer provee un registro global static, para acceder a el puedes utilizar:

Metrics.globalRegistry que es de tipo **CompositeMeterRegistry**

Medidas (Meters)

Micrometer viene con un conjunto de medidas soportadas entre las que se encuentran:

- **Timer**
- **Counter**
- **Gauge**
- **DistributionSummary**
- **LongTaskTimer**
- **FunctionCounter**
- **TimeGauge**

Diferentes tipos de medidas resultan en diferentes numeros de series.

Una medida es identificada por su nombre y dimensión. Se utiliza el término dimensión y tags de forma indistinta. Dimensiones permiten a un nombre de métrica en particular ser partida para analizar los datos a profundidad.

Nombrado de métricas

Micrometer emplea una convención de nombres que utiliza minúsculas separadas con un '.' punto. Diferentes sistemas de monitoreo tienen diferentes sistemas de nombrado por lo que cada implementación de micrometer viene con un transformador a la estructura recomendada, puedes sobreescribir la convención de nombres por defaul implementando la interfaz **NamingConvention**, como se muestra a continuación:

```
meterRegistry.config().namingConvention(...);
```

Counter

Un Counter reporta simplemente un contador sobre la propiedad específica de una aplicación. a continuación un ejemplo:

```
meterRegistry.counter("devs4j.students",
"profile","frontend").increment();

meterRegistry.counter("devs4j.students",
"profile","backend").increment(5);
```

Timers

Para medir latencias o frecuencia de eventos puedes utilizar Timers. a continuación un ejemplo:

```
SimpleMeterRegistry registry = new
SimpleMeterRegistry();
Timer timer =
registry.timer("devs4j.processing.time");
timer.record( ()->{
        try {
                Thread.sleep(new
Random().nextInt(1500));
        } catch (InterruptedException e) {}
        System.out.println("Waiting...");
});
log.info("Total time {}
ms",timer.totalTime(TimeUnit.MILLISECONDS));
```

Gauge

Un Gauge muestra el valor actual de una medida, son utilizados para monitorear estadísticas de cache, colecciones, etc:

```
SimpleMeterRegistry registry = new SimpleMeterRegistry();

List<String> list = new ArrayList<>(4);

Gauge gauge = Gauge.builder("list.size", list,
List::size).register(registry);

log.info("Value {}", gauge.value());

list.addAll(
Arrays.asList("@devs4j","@raidentrance","@springframework"));

log.info("Value {}", gauge.value());
```

La salida del código anterior mostrará:

Value : 0.0
Value : 3.0

DistributionSummary

Un **DistributionSummary** es utilizado para dar seguimiento a la distribución de eventos. Es similar a un **Timer** de forma estructural, pero los registros no representan una unidad de tiempo. Por ejemplo un **distribution summary** puede ser utilizado para medir el tamaño de la respuesta de las peticiones que recibe un servidor, a continuación un ejemplo:

```
DistributionSummary summary = registry.summary("salary.summary",
"job", faker.job().position());

summary.record(salary);
```

Configuración

Todas las clases necesarias para trabajar con micrometer las tendremos al incluir la dependencia:

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-core</artifactId>
     <version>1.7.1</version>
</dependency>
```

Si se trabajará con Spring Framework + Prometheus como es en el caso de este curso se puede incluir solo la dependencia:

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

<plugin>
<artifactId>maven-compiler-plugin</artifactId>
<version>3.7.0</version>
 <configuration>
    <source>1.8</source>
    <target>1.8</target>
 </configuration>
</plugin>
```

DEVS4J
Hecho por Devs para Devs

Métricas y micrometer

@Timed

Puedes anotar un método con **@Timed** para medir el tiempo de ejecución de un método definido en un **@Controller**. Al agregar la anotación se generarán 2 series con los siguientes nombres:

-`${name}_count`- Número total de todas las peticiones
-`${name}_sum`- Tiempo total de todas las peticiones

Puedes calcular el tiempo promedio con el siguiente query de Prometheus:

```
rate(timer_sum[10s])/rate(timer_count[10s])
```

Puedes calcular el **throughput**(Peticiones por segundo) como se muestra a continuación:

```
rate(timer_count[10s])
```

NOTA: No es posible utilizar @Timed fuera del contexto web en un método regular.

Long task timer

Los long task timers son un tipo especial de timers que permiten medir el tiempo de un evento que sigue en ejecución. Un timer regular no calcula la duración hasta que el proceso se completa.

En una aplicación de Spring, es común ejecutar procesos largos con la anotación **@Scheduled**, puedes aplicar un long task timer como se muestra a continuación:

```
@Timed(value = "long_time", longTask = true)
@Scheduled(fixedDelay = 360000)
void longTimeProcess() {

}
```

Puede ser utilizado para notificar si el tiempo de ejecución supera algún threshold definido.

Puedes calcular la duración con :

```
longTaskTimer{statistic="duration"}
```

Prometheus

Las méticas se pueden consultar siguiendo la estructura de prometheus, para esto consultaremos utilizando la siguiente url:

/actuator/prometheus

Hecho lo anterior veremos una salida como la siguiente:

```
# HELP get_characters_seconds   # TYPE
get_characters_seconds summary
get_characters_seconds_count
{exception="None",method="GET",
status="200",uri="/characters",} 1.0
get_characters_seconds_sum
{exception="None",method="GET",
status="200",uri="/characters",} 0.009020125
```

Descarga de prometheus

Puedes descargar prometheus del siguiente enlace:

https://prometheus.io/download/

Una vez descargado, lo debes descomprimir entrar a la carpeta y ejecutar:

```
./prometheus --config.file=prometheus.yml
```

Una vez iniciado puedes acceder a la siguiente URL:

http://localhost:9090/graph

Modificaremos el archivo prometheus.yml para que utilice las métricas que estamos generando en nuestra aplicación como se muestra a continuación:

```
scrape_configs:
  - job_name: 'prometheus'
    metrics_path: /actuator/prometheus

    scrape_interval: 5s
    static_configs:
      - targets: ['localhost:8080']
```

Una vez hecho lo anterior entra de nuevo a la Url y verás las métricas de tu aplicación

Descarga grafana

Puedes descargar grafana en :

https://grafana.com/grafana/download

Una vez descargado entra al folder en la carpeta de bin y ejecuta:

```
./grafana-server
```

Y abre la URL http://localhost:3000/

Selecciona en el side bar:

- Configuración
- Datasources
- Add datasource
- Selecciona Prometheus
  - -Name = Prometheus
  - -Url = http://localhost:9000

Creación de un dashboard

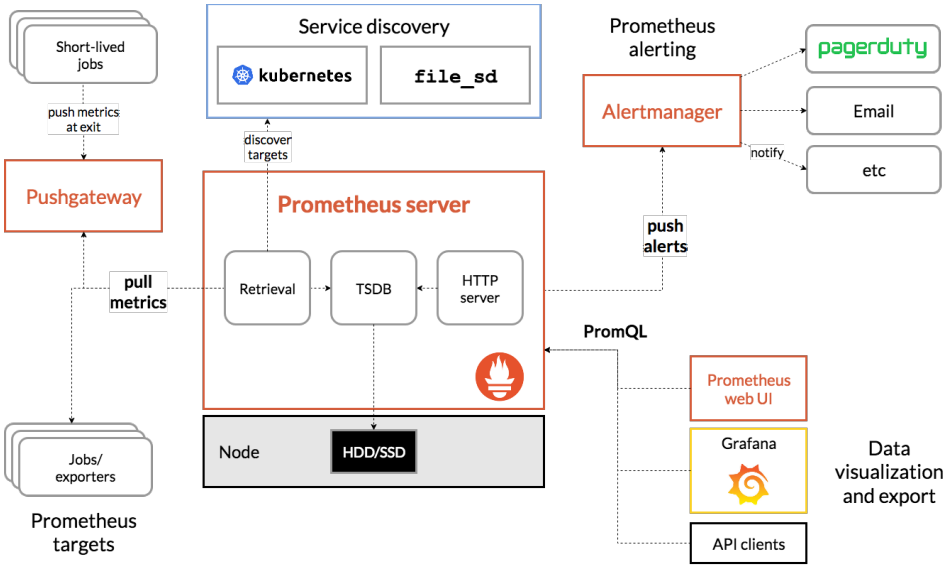
Para crear un dashboard de grafana selecciona el símbolo + de la barra de la izquierda y sigue los siguientes pasos:

- Crea un panel

- En la sección de Metrics, selecciona Prometheus y coloca el query que deseas utilizar, en el ejemplo utilizaremos :

`rate(get_characters_seconds_count[5m])`

- En el panel de la derecha puedes definir el título y el tipo de gráfica que deseas representar



DEVS4J
Hecho por Devs para Devs

www.twitter.com/devs4j

www.facebook.com/devs4j

www.devs4j.com

Monitoring and Alerting