WORD EEN SUPERHELD MET MICROMETER

Wist je dat veel calls van klanten voorkomen kunnen worden als ze gebruik maken van onze software? Door eerder te acteren op basis van juiste monitoring, kunnen we onze klanten beter en eerder helpen! Niet met logging, maar met het maken van (real-time) metrics in je code.

Ko Turk, Java Developer Blue4IT bij de Rabobank.

{ WAT ZIJN METRICS? }

Zie het naar de dokter gaan als je ziek bent. Je kan aangeven waar je bent geweest, naar feestjes, festivals of dergelijke. Dat noemen we logging. Een metric kunnen we vergelijken met bijvoorbeeld het controleren van je lichaamstemperatuur of het onderzoeken van je bloed. Het geeft je veel meer inzicht. En hoe eerder je het weet, hoe sneller je beter wordt. Dit geldt ook voor je applicatie! Micrometer is een library die ons kan helpen deze metrics te maken.

{ HOE BEGIN IK? }

Maak je gebruik van Spring Boot, Micronaut of Quarkus? Dan heb ik goed nieuws! Ook jij kan Micrometer toevoegen aan je project.

Voor Spring Boot voeg je de volgende Maven dependency toe aan je pom.xml:

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

Nog even je application.yml updaten met de volgende waarde:

```
   management:
      endpoints:
         web:
           exposure:
             include: 'metrics'
```

En klaar ben je om je eigen metrics te creëren! Je krijgt standaard al een aantal 'gratis' metrics, zoals informatie over het memory-gebruik, de garbage collector, aantal threads, buffers en klasses binnen je JVM. Wanneer je de Spring Boot applicatie start, kan je deze metrics vinden onder:

```
/actuator/metrics
```

Iedere afzonderlijke metric kun je aanroepen door de naam achter de metrics endpoint toe te voegen.

Gebruik je geen Spring Boot, maar Micronaut? Hiervoor voeg je twee dependencies toe, zoals `io.micronaut:micronaut-micrometer-core` en `io.micronatu:micronaut-management`.

Voor Quarkus gebruik je `io.quarkus:io.quarkus:quarkus-micrometer`.

{ WAT WIL JE GAAN MONITOREN? }

Voordat je begint met het maken van metrics, is het belangrijk om na te gaan wat je wil monitoren. Heb je bepaalde afspraken gemaakt met je klant met betrekking tot snelheid of uptime? Dan is het wellicht handig om hier metrics bij te bedenken. We gaan nu verder in op enkele use cases bij het maken van metrics.

{ PERFORMANCE MONITOREN MET TIMERS }

Wil je graag de performance van je eigen microservice(s) monitoren? Dan is de Timed annotatie echt iets voor jou. Het enige wat je hoeft te doen, is de annotatie boven je methode te zetten en een naam te geven, en klaar ben je!

```
@Timed("endpoint.timer")
```

Heb je te maken met calls naar andere systemen? Zoals een aan-

roep naar een microservice, database, of wellicht een heel oud systeem dat nog werkt met SOAP? Dan is de annotatie wellicht niet de beste keus. Je kunt ook een timer metric binnen je methode plaatsen, deze heeft een `record` functie. Let er wel op dat deze een duration verwacht, dus zorg zelf voor de start- en eindtijd.

```
var duration = Duration.between(beginTimestamp,
endTimestamp);
Metrics.timer("method.timer").record(duration);
```

Voor migratiewerkzaamheden kan het ook handig zijn om de tijd over een langere periode te bewaken. Het kan namelijk handig zijn om te weten hoe lang een actie al loopt. Daar gebruiken we een `longtask` timer voor.

```
@Timed(value = "migration.timer", longTask = true)
```

{ BERICHTEN MONITOREN MET COUNTERS }

Werk je met event streaming applicaties? Dan kan het handig zijn om het aantal berichten te tellen dat binnenkomt. Hier kun je bijvoorbeeld ook nog een bepaalde waarde aan meegeven, een zogenoemde 'tag' (meer daarover later in dit artikel). Een counter kan overigens alleen optellen en kan niet negatief zijn.

```
Metrics.counter("message.counter").increment();
```

Er zijn natuurlijk een aantal andere use cases te bedenken om een counter te maken. Wat dacht je bijvoorbeeld van een order counter?
Gebruik je Spring Kafka en maak je gebruik van consumers met annotatie `@KafkaListener`? Dan krijg je zelfs al een aantal gratis metrics cadeau. Van connections, latency tot aan heartbeats, je vindt het er allemaal.

{ COLLECTIES MONITOREN MET GAUGES }

Soms kan het ook van betekenis zijn om een waarde uit je collectie te monitoren. Bijvoorbeeld de size van een List, Map of Set. Of wat dacht je van het monitoren van je queues, mocht je organisatie nog MQ gebruiken? En ook als je gebruik maakt van caches, zoals Hazelcast, EHCache of iets anders, is het handig om te weten wat de inhoud hiervan is.

```
Metrics.gauge("queue.size", queue.size());
```

{ PRIORITEITEN ACHTERHALEN MET DE EXCEPTION TAG }

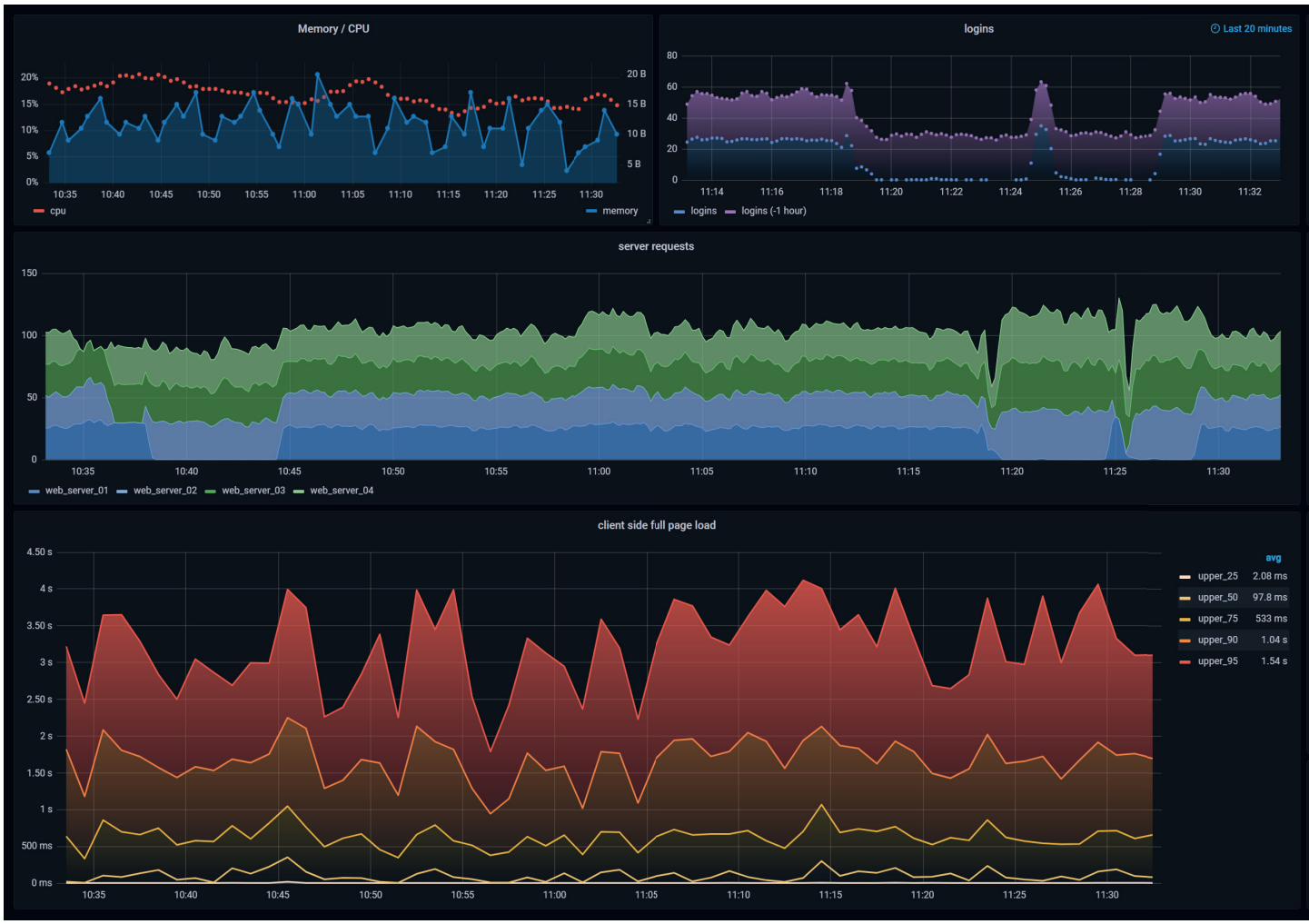
Heb je een Product Owner die graag zoekt naar prioriteiten van een bepaald incident? Binnen de metrics (Timer, Counter en Gauge) heb je ook een tag `exception`. Hiermee is het mogelijk om te zien of er excepties worden gegooid, welke niet worden afgevangen. Als deze waarde heel hoog staat, weet je dat je een groot probleem hebt in je software en moet je actie ondernemen. Een voorbeeld van het uitlezen van een specifieke exception binnen je applicatie:

```
/actuator/metrics/http.server.
requests?tag=exception:SomeException
```

Deze metric geeft een counter terug met het aantal keren dat deze exceptie is voorgekomen. Binnen deze counter heb je ook de mogelijkheid om te zien welke statuscode wordt gegooid. In dit geval een 500 en ook zie je op welke URI dit was.

{ TIMER MAKEN VOOR VERSCHILLENDE SYSTEMEN }

Een ander voorbeeld voor het maken van een tag. Je wil bijvoorbeeld de tijd naar verschillende systemen monitoren. Je kan dan een tag meegeven zoals `systeem` en daaraan een waarde koppe-



len als `SysteemA`, `SysteemB` of `SysteemC`. Met deze manier kun je ook nog eens alle waarden met elkaar vergelijken.

```
Metrics.timer("method.timer", "systeem",
"SysteemA").record(duration);
```

Een tag kan je aanmaken voor een Timer, Counter of een Gauge.

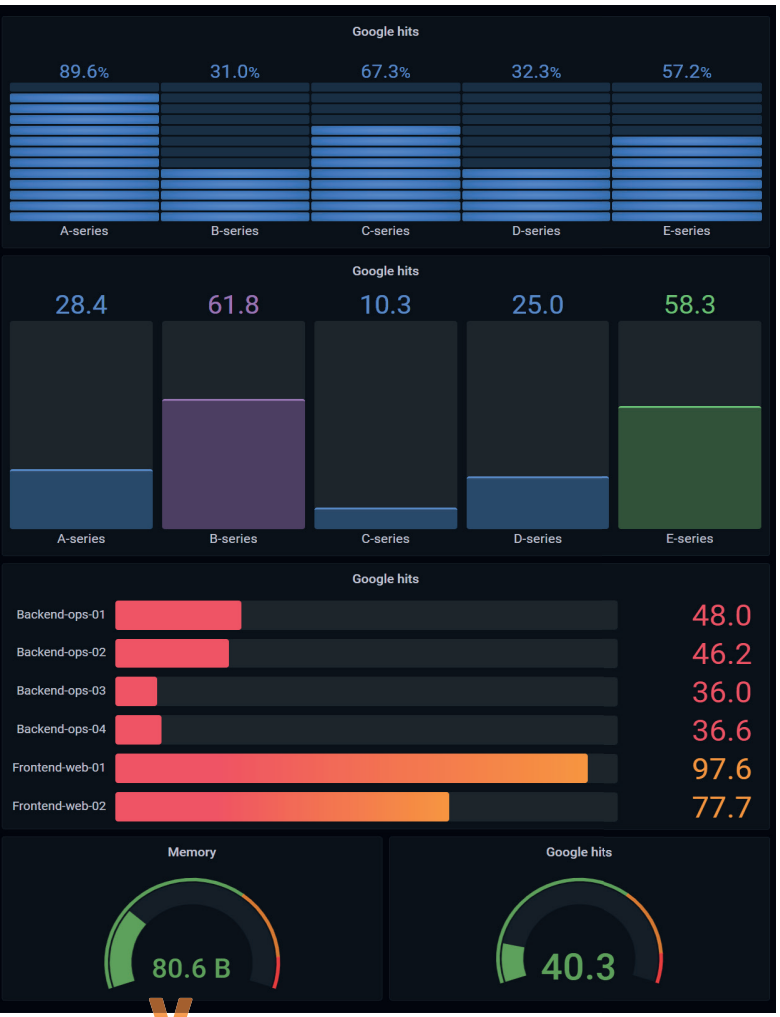
{ DENK AAN SECURITY(!) }

Voordat je alles commit, pushed, released en deployed is het goed om te weten dat de endpoints momenteel openstaan voor iedereen die zich in je netwerk bevindt. Met het oog op de AVG/GDPR is het verstandig om de endpoints te beveiligen of dat je de mogelijkheid uitzet waarmee gebruikers bij je applicatie kunnen komen.

Het beveiligen kan dus op twee manieren. Eén daarvan is het implementeren van Spring Security. Je stelt de endpoints dan wel open, maar alleen voor mensen met een gebruikersnaam en wachtwoord.

Je kunt er ook voor kiezen om je gegevens elders op te slaan. Gebruik dan de pushmogelijkheden binnen Spring Actuator (daarover later meer). Per default zijn de endpoints `/health` en `/info` toegankelijk. In oudere versies van Actuator moest je deze specifiek uitzetten, maar dat hoeft dus niet meer. Wil je toch meer endpoints openzetten? Dan kan dat ook. Maar gebruik geen `'*'` in je include zonder na te denken en zonder te beveiligen. Voorbeeld van de application.yml:

```
management:
 endpoints:
   web:
     exposure:
       include: 'health, metrics'
# mocht je een oudere versie van Spring Actuator
gebruiken, voeg dan het
# volgende toe
   jmx:
     exposure:
       exclude: '*'
```



Een voorbeeld van een Grafana dashboard, link: **https://play.grafana.org/d/3SWXxreWk/.**

{ HET OPSLAAN VAN DATA }

Zoals aangegeven zijn er ook mogelijkheden om je data elders op te slaan. Er zijn hierbij twee mogelijkheden om je informatie te ontsluiten. Aan de ene kant dus via push, gebruik dan systemen als DataDog, SignalFX, Graphite of een andere ondersteunende service. En aan de andere kant heb je pullsystemen als Prometheus, maar die verwachten een endpoint om uit te lezen. Het enige wat je in je code hoeft te doen, is een dependency toe te voegen:

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

En in je application.yml moet je de endpoint exposen:

```
management:
 endpoints:
   web:
     exposure:
       include: 'prometheus'
```

Door Prometheus te enablen krijg je een nieuwe endpoint (mits je hem niet beschikbaar stelt). Deze lijkt op de metrics endpoint, maar heeft een ander format met extra informatie.
Ga je migreren naar een ander pakket? Dan hoef je alleen het voorgaande te veranderen.

{ EDGE CASES VINDEN BINNEN EEN TIMER }

Door gebruik te maken van Prometheus krijg je overigens wel extra functionaliteiten. Als je timers wil aanmaken, krijg je ook de beschikking over histogrammen. Dit houdt in dat je apart bakjes (buckets) kunt bijhouden. Dat kan bijzonder handig zijn als je alleen een bepaald spectrum (of edge cases) wil onderzoeken. Je kan dan een bakje maken van bijvoorbeeld de calls die onder de SLA-tijd blijven. Je kan daarnaast ook nog bakjes maken van de minimale verwachte tijd en de maximale tijd (gebaseerd op percentielen).

```
Timer timer = Timer
        .builder("sla.timer")
        .publishPercentiles(0.3, 0.5, 0.95)
        .publishPercentileHistogram()
        .sla(Duration.ofSeconds(4))
        .minimumExpectedValue(Duration.
        ofSeconds(1))
        .maximumExpectedValue(Duration.
        ofSeconds(3))
        .register(registry);
```

Zoals je ziet, verwacht de timer een `registry`. Declareer deze ook binnen je klasse:

```
MeterRegistry registry;
```

{ HET VISUALISEREN VAN JE METRICS }

En nu komt het laatste gave punt! Je kan ook een dashboard koppelen aan je database, zoals Grafana, Kibana, Dynatrace of één van de alleskunners, zoals SignalFX en DataDog.
Er zijn wederom meerdere wegen naar Rome en dus ook verschillende systemen die ons hierbij kunnen helpen. Binnen deze systemen kun je de metrics visualiseren met grafieken, tabellen en diagrammen. Op basis van de data kan je daarnaast ook alerts instellen. Dus als er iets misgaat, ben je snel op de hoogte. Zo kunnen we onze klanten sneller bedienen en behoeden tegen bugs en outage.
Wil je meer weten over Micrometer? Kijk dan eens op **https://micrometer.io/**