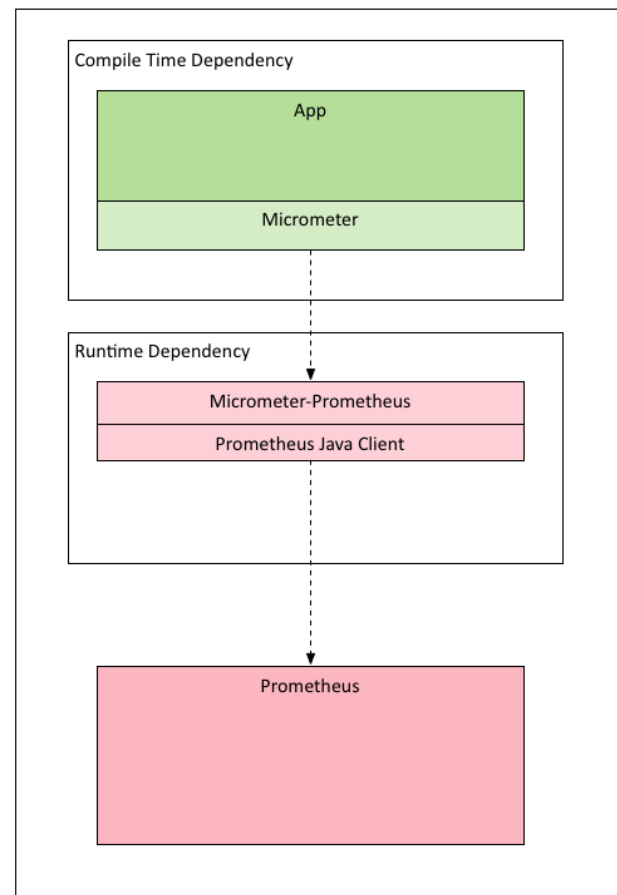


Prometheus, Grafana, Micrometer

<https://github.com/jirkapinkas/javadays-2020>

Úvod

- Spring Boot Actuator umožňuje zpřístupnění metrik. Existuje řada serverů, které tyto metriky sbírají (Prometheus, Elastic, DataDog, Dynatrace, New Relic atd.), přičemž každý z těchto serverů vyžaduje jejich export ze serveru v určitém formátu.
- Spring Boot pro export metrik používá knihovnu Micrometer, která funguje jako fasáda mezi aplikací a serverem, který tyto metriky sbírá (jako SLF4J pro logy).
- V této přednášce se podíváme na integraci se serverem Prometheus, který slouží pro sběr metrik a další nástroj (Grafana), který slouží k jejich vizualizaci.



Základní konfigurace

- Minimálně potřebujeme tyto dependency (pom.xml):

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

- A následující konfiguraci (application.properties):

```
management.endpoints.web.exposure.include=metrics,prometheus
```

- Poté budou fungovat URL:
 - <http://localhost:8080/actuator/metrics>
 - <http://localhost:8080/actuator/prometheus>

Pokročilejší konfigurace

- Pokud chcete z Actuatoru, Prometheus & Grafany dostat maximum, pak bych doporučil rozšířit konfiguraci následovně (application.properties):

```
# díky tomuto nastavení fungují snad všechny dashboardy https://grafana.com/grafana/dashboards/4701  
spring.application.name=app  
management.metrics.tags.application=${spring.application.name}
```

```
# nastavení pro: https://grafana.com/grafana/dashboards/9845  
management.metrics.web.server.request.autotime.percentiles=0.5,0.9,0.95,0.99  
management.metrics.web.server.request.autotime.percentiles-histogram=true
```

```
# nastavení pro: https://grafana.com/grafana/dashboards/12464  
server.tomcat.mbeanregistry.enabled=true  
spring.jpa.properties.hibernate.generate_statistics=true
```

Základní metrika

- Poté například pro metriku:
 - <http://localhost:8080/actuator/metrics/process.threads>

- Bude výpis z prometheus endpointu:

```
# HELP process_threads The number of process threads
```

```
# TYPE process_threads gauge
```

```
process_threads{application="app",} 44.0
```

Tag, který říká
z jaké aplikace
tato metrika pochází

Hodnota

Typ Prometheus metriky,
Gauge = hodnota, která může jít nahoru nebo dolů.
Odpovídá measurements.statistic = VALUE

```
{  
  "name": "process.threads",  
  "description": "The number of process  
    threads",  
  "baseUnit": null,  
  "measurements": [ {  
    "statistic": "VALUE",  
    "value": 42  
  }  
],  
  "availableTags": [ {  
    "tag": "application",  
    "values": [  
      "app"  
    ]  
  }  
]
```

Metrika s větším množstvím tagů I.

- Tato metrika má větší množství tagů:
 - <http://localhost:8080/actuator/metrics/logback.events>

```
{
  "name": "logback.events",
  "description": "Number of error level events that made it to the logs",
  "baseUnit": "events",
  "measurements": [ {
    "statistic": "COUNT",
    "value": 57
  } ],
  "availableTags": [ {
    "tag": "application",
    "values": [ "app" ]
  }, {
    "tag": "level",
    "values": [ "warn", "trace", "debug", "error", "info" ]
  } ]
}
```

Sumární informace, počet všech log hlášek

Tagy

Metrika s větším množstvím tagů II.

- Jak získat detail takové metriky? V tomto případě informaci, kolik bylo například “warn” hlášek?

- <http://localhost:8080/actuator/metrics/logback.events?tag=level:warn>

Opět typ Prometheus metriky, Counter = hodnota se v čase incrementuje

```
{
  "name": "logback.events",
  "description": "Number of warn level events that made it to the logs",
  "baseUnit": "events",
  "measurements": [ {
    "statistic": "COUNT",
    "value": 1
  }
],
  "availableTags": [ {
    "tag": "application",
    "values": [ "app" ]
  } ]
}
```


To samé v Prometheus formátu:

```
# HELP logback_events_total Number of error level
#       events that made it to the logs
# TYPE logback_events_total counter
logback_events_total{application="app",level="trace",} 0.0
logback_events_total{application="app",level="info",} 56.0
logback_events_total{application="app",level="warn",} 1.0
logback_events_total{application="app",level="error",} 0.0
logback_events_total{application="app",level="debug",} 0.0
```

Prometheus server: konfigurace

docker-compose.yml:

```
services:
  app:
    container_name: app
    build: ./spring-boot-prometheus
    ports:
      - "8080:8080"
  prometheus:
    container_name: prometheus
    image:
      prom/prometheus
    ports:
      - "9090:9090"
    volumes:
      - ./volumes/prometheus.yml:/etc/prometheus/prometheus.yml
```

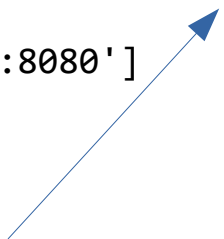



Prometheus bude běžet na:
<http://localhost:9090>

volumes/prometheus.yml:

```
global:
  scrape_interval: 10s
  evaluation_interval: 30s
  external_labels:
    monitor: 'codelab-monitor'

scrape_configs:
  - job_name: 'app'
    metrics_path: '/actuator/prometheus'
    scrape_interval: 5s
    static_configs:
      - targets: ['app:8080']
```



Prometheus bude scrapovat
<http://app:8080/actuator/prometheus> každých 5 vteřin

http://localhost:9090/targets

Prometheus Alerts Graph Status ▾ Help

Targets

All Unhealthy Collapse All

app (1/1 up) [show less](#)

Runtime & Build Information

Command-Line Flags

Configuration

Rules

Targets

Service Discovery

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://app:8080/actuator/prometheus	UP	<code>instance="app:8080"</code> <code>job="app"</code>	3.203s ago	88.98ms	

up

- Když Prometheus provádí scrapování, tak uloží label “up” (a pár dalších). Up nabývá hodnot: 0, 1 (nula když scrapování selhalo, jedna když se podařilo).
- Toto je užitečné pro monitoring, jestli aplikace vůbec žije (a jestli monitoring této aplikace vůbec funguje).
 - https://prometheus.io/docs/concepts/jobs_instances/

Příklady

- `http_server_requests_seconds_count`
 - Celkový počet požadavků na server
- `http_server_requests_seconds_sum`
 - Suma délky trvání všech requestů na server
- `http_server_requests_seconds_count[5m]`
 - Všechny počty požadavků na server za posledních 5 minut
- `rate(http_server_requests_seconds_count[5m])`
 - Průměrný počet požadavků na server za vteřinu za posledních 5 minut
- `rate(http_server_requests_seconds_sum[5m])/rate(http_server_requests_seconds_count[5m])`
 - Průměrná délka odezvy serveru za posledních 5 minut
- `rate(http_server_requests_seconds_sum{status="200"}[5m])/rate(http_server_requests_seconds_count{status="200"}[5m])`
 - Průměrná délka odezvy serveru u požadavků, jejichž status kód byl 200 za posledních 5 minut

http://localhost:9090/graph (Console)

Prometheus Alerts Graph Status ▾ Help

☐ Enable query history

[Try experimental React UI](#)

```
rate(http_server_requests_seconds_sum{status="200"}[1m])/rate(http_server_requests_seconds_count{status="200"}[1m])
```

Load time: 56ms
Resolution: 14s
Total time series: 1

Execute

- insert metric at cursor - ▾

[Remove Graph](#)

Graph

Console



Moment



Element	Value
{application="app",exception="None",instance="app:8080",job="app",method="GET",outcome="SUCCESS",status="200",uri="/actuator/prometheus"}	0.08306919999999997

http://localhost:9090/graph (Graph)



Typy Prometheus metrik

- Prometheus má 4 základní metriky:
 - COUNTER
 - Cokoli, co se v čase pouze incrementuje
 - Například počet zpráv v logu
 - GAUGE
 - Hodnota, která může jít nahoru i dolů
 - Například množství použité paměti
 - HISTOGRAM
 - Hodnota, která je v nějakém intervalu hodnot
 - Například jaká byla průměrná odezva 90% requestů
 - SUMMARY
 - Podobné jako histogramy, ale výpočet se provádí na klientovi, nikoli na Prometheus serveru

Vlastní Gauge & Counter metrika

@Component

```
public class CustomMeterBinder implements MeterBinder {
```

```
    private Counter counter;
```

@Override

```
public void bindTo(MeterRegistry meterRegistry) {  
    Gauge.builder("custom_gauge", this, value -> value.getCustomValue())  
        .description("Custom Gauge")  
        .register(meterRegistry);  
    counter = Counter.builder("custom_counter")  
        .description("Custom Counter")  
        .register(meterRegistry);  
}
```

```
private Double getCustomValue() {  
    counter.increment();  
    return (double) new Random().nextInt(10);  
}
```

```
}
```

```
# HELP custom_gauge Custom gauge  
# TYPE custom_gauge gauge  
custom_gauge{application="app",} 5.0  
# HELP custom_counter_total Custom Counter  
# TYPE custom_counter_total counter  
custom_counter_total{application="app",} 25.0
```

Vlastní Timer metrika

Micrometer má možnost lehce vytvořit “Timer” metriku (vytvoří GAUGE a SUMMARY):

```
@Component
public class CustomMeterBinder2 implements MeterBinder {

    private Timer timer;

    @Override
    public void bindTo(MeterRegistry meterRegistry) {
        timer = meterRegistry.timer("long.operation.run.timer");
    }

    @Scheduled(fixedDelay = 3_000)
    public void sampleLongOperation() throws InterruptedException {
        long startTime = System.nanoTime();
        Thread.sleep(new Random().nextInt(7_000));
        timer.record(System.nanoTime() - startTime, TimeUnit.NANOSECONDS);
    }
}
```

```
# HELP long_operation_run_timer_seconds
# TYPE long_operation_run_timer_seconds summary
long_operation_run_timer_seconds_count{application="app",} 24.0
long_operation_run_timer_seconds_sum{application="app",} 89.7692185
# HELP long_operation_run_timer_seconds_max
# TYPE long_operation_run_timer_seconds_max gauge
long_operation_run_timer_seconds_max{application="app",} 6.4372651
```


@Timed

@Configuration

```
public class TimedConfiguration {
```

@Bean

```
public TimedAspect timedAspect(MeterRegistry registry) {  
    return new TimedAspect(registry);  
}
```

```
}
```

@RestController

```
public class HelloController {
```

@Timed

```
@GetMapping("/message")  
public Message message() {  
    return new Message("stuff");  
}
```

```
}
```

Úplně to samé se dá udělat pomocí anotace @Timed, jenom se musí jednorázově zapnout podpora pro tuto funkcionalitu.

```
# HELP method_timed_seconds  
# TYPE method_timed_seconds summary  
method_timed_seconds_count{application="app",  
class="com.example.springbootprometheus.controller.HelloController",  
exception="none",method="message",} 3.0  
method_timed_seconds_sum{application="app",  
class="com.example.springbootprometheus.controller.HelloController",  
exception="none",method="message",} 0.0102852  
# HELP method_timed_seconds_max  
# TYPE method_timed_seconds_max gauge  
method_timed_seconds_max{application="app",  
class="com.example.springbootprometheus.controller.HelloController",  
exception="none",method="message",} 0.0101703
```

Grafana

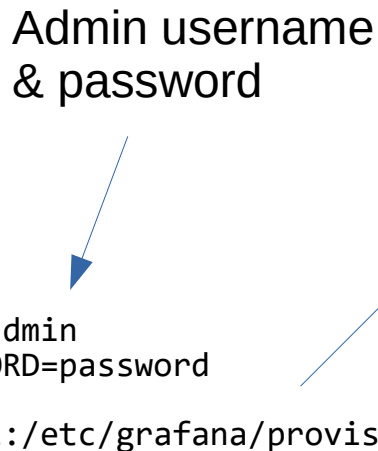
- Prometheus umožňuje základní vizualizaci PromQL výrazu a je výborný pro odladění nějakého výrazu, ale pro vizualizaci metrik a tvorbu dashboardů, kde bude na jednom místě více komponent zobrazujících aktuální stav systému se obvykle používá Grafana.
- Zde jsou navíc externalizované volumes aby se při rebuildu Promethea a Grafany nepřišlo o data, to jsem do příkladu nedal:
 - <https://dimitr.im/monitoring-spring-prometheus-grafana>

Grafana server: konfigurace

docker-compose.yml:

```
grafana:
  container_name: grafana
  image:
    grafana/grafana
  ports:
    - "3000:3000"
  environment:
    - GF_SECURITY_ADMIN_USER=admin
    - GF_SECURITY_ADMIN_PASSWORD=password
  volumes:
    - ./volumes/datasource.yml:/etc/grafana/provisioning/datasources/datasource.yml
```

Admin username
& password



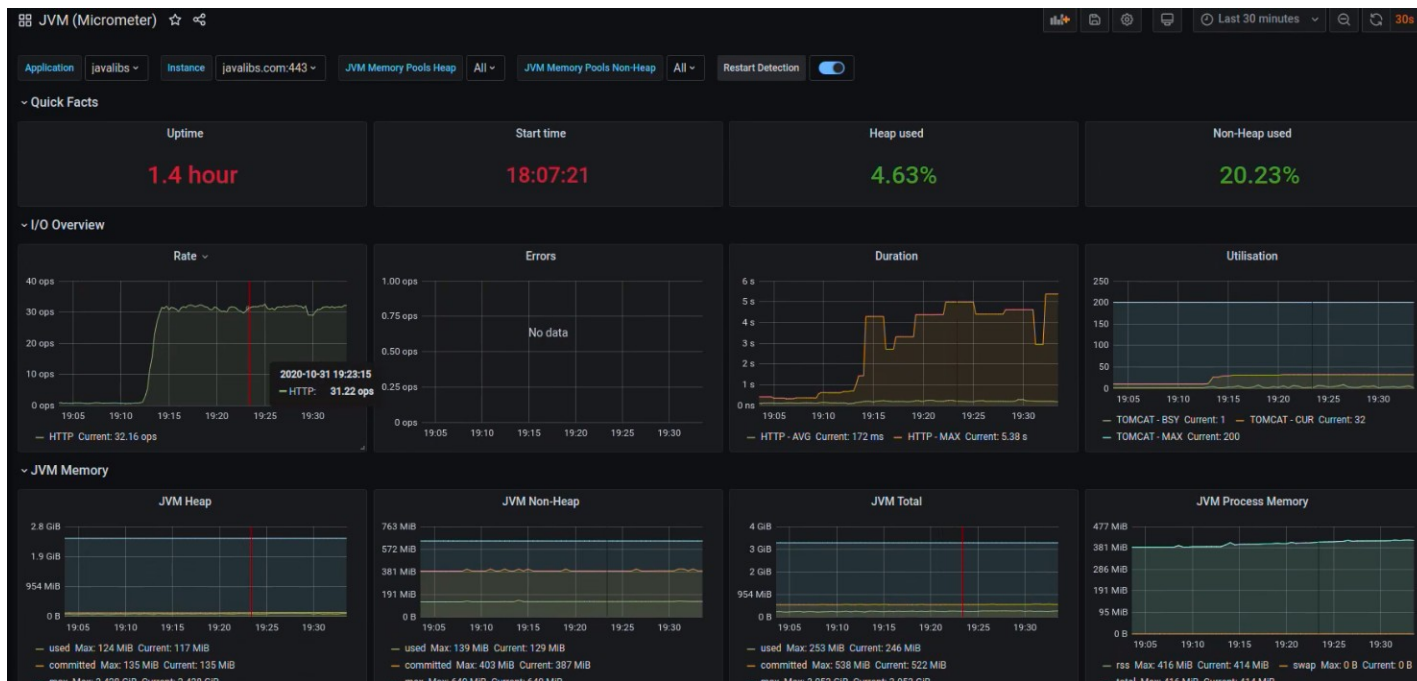
volumes/datasource.yml:

```
apiVersion: 1
datasources:
  - name: Prometheus
    type: prometheus
    access: proxy
    url: http://prometheus:9090
```

Grafana bude běžet na:
<http://localhost:3000>

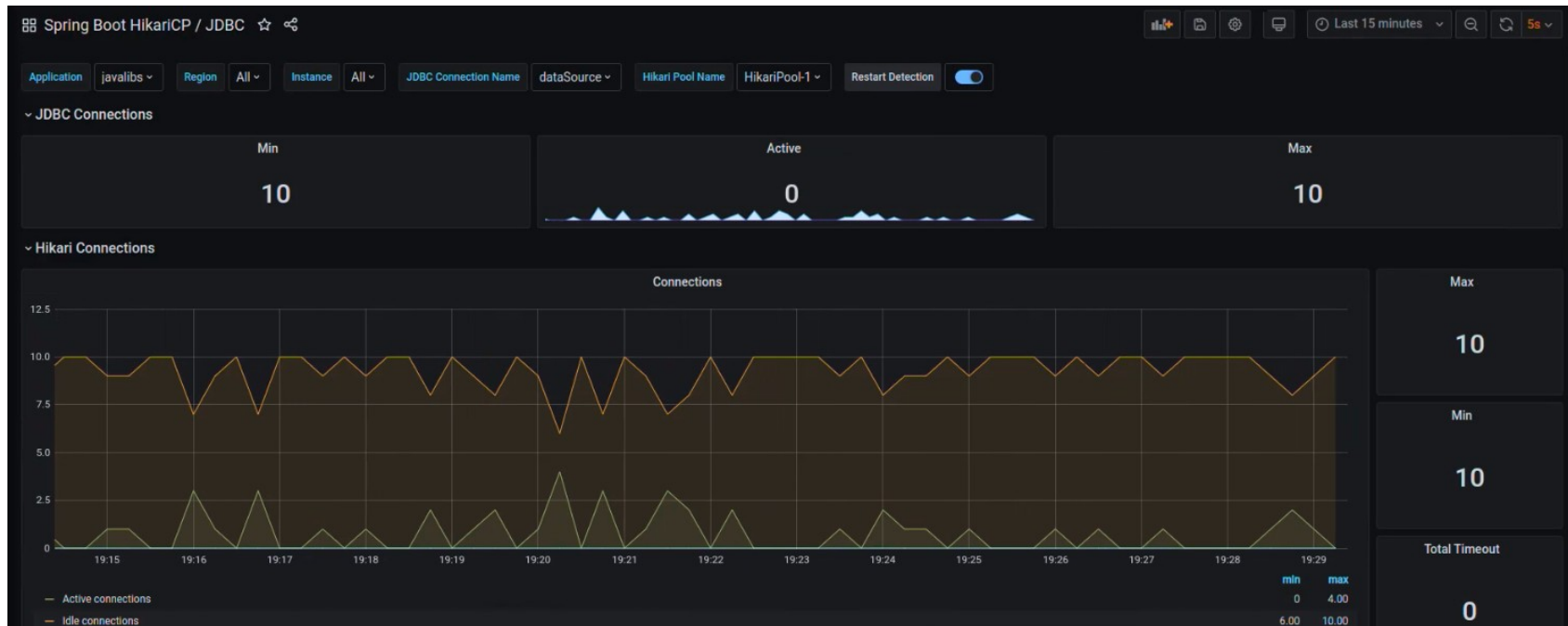
Micrometer Dashboard

- Velice pěkný Grafana dashboard pro Micrometer (pouze Hotspot), funguje out-of-the-box (4701) a obsahuje základní metriky (paměť, CPU, load apod.):
 - <https://grafana.com/grafana/dashboards/4701>



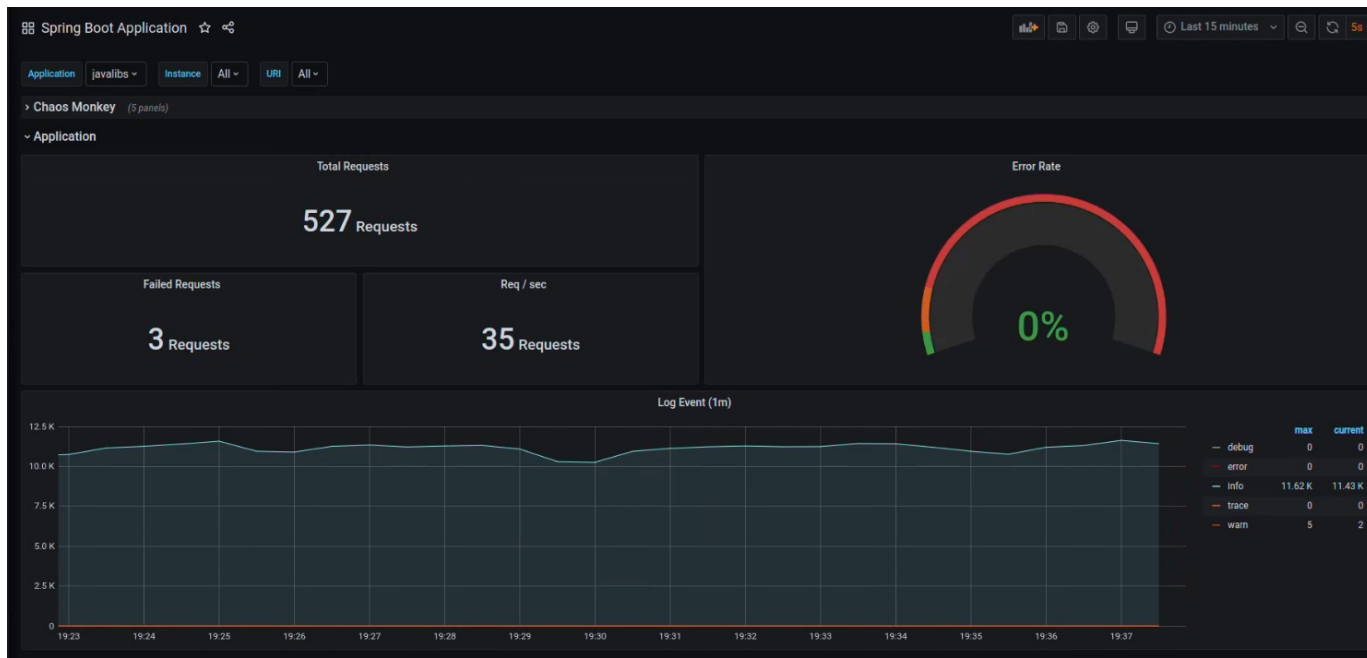
Spring Boot HikariCP / JDBC

- Další dashboard, který funguje out-of-the-box a je zaměřený na monitorování stavu connection poolu (6083):
 - <https://grafana.com/grafana/dashboards/6083>



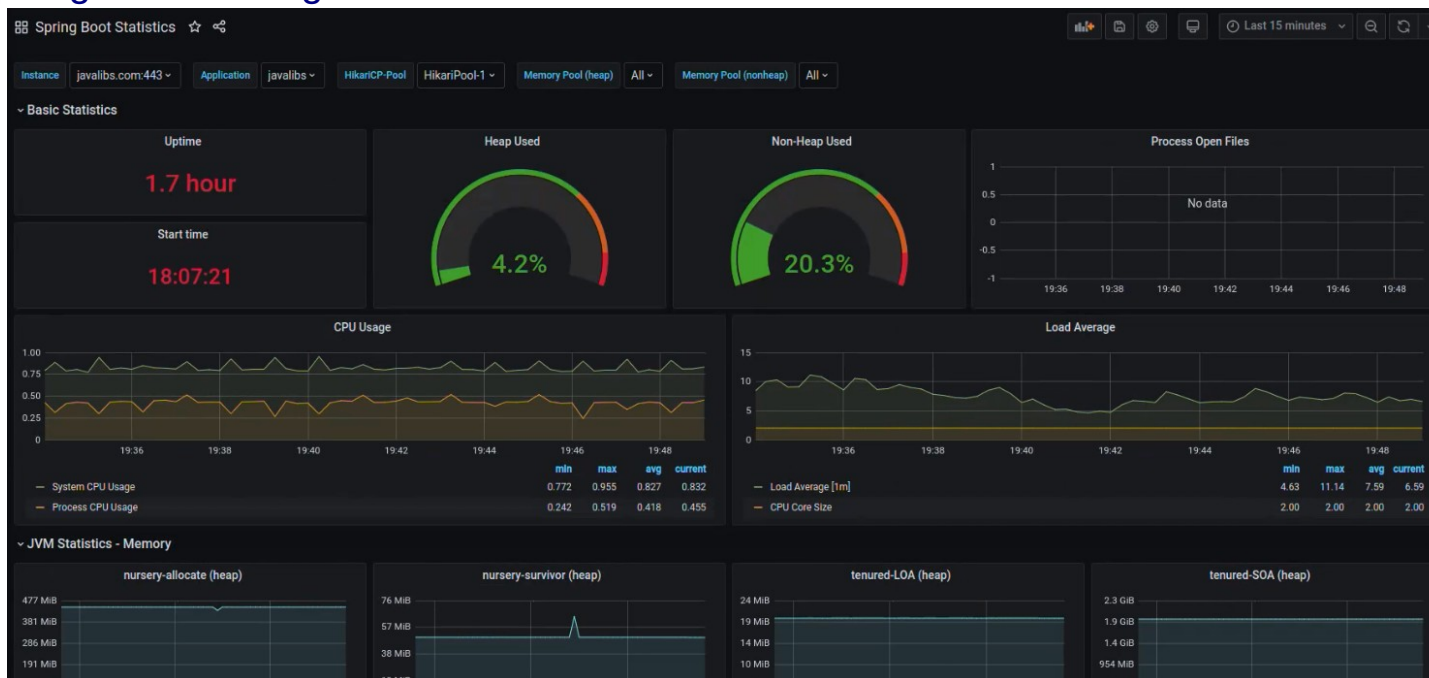
Spring Boot Application

- V tomto dashboardu v některých grafech nefunguje URI, ale když se odstraní, tak vše funguje jak má. Také se v něm může zobrazovat response time RestTemplate / OkHttp požadavků (u microservice architektury) ... když se to nastaví (9845).
 - <https://grafana.com/grafana/dashboards/9845>



Spring Boot Statistics (Jetty)

- Tento dashboard (12464) vyžaduje trošku nastavení, ale odmění se hromadou velice užitečných statistik. V nastavení dashboardu se změní u proměnné “application”: `jvm_classes_loaded` → `jvm_classes_loaded_classes` (původní label byl ve Spring Boot 2.0, od 2.1 je to jinak):
 - <https://grafana.com/grafana/dashboards/12464>



Spring Boot Statistics (Tomcat)

- Předchozí dashboard (12464) je forkem dashboardu s číslem 6756. V původním dashboardu byly informace o tom, jak hodně se používá thread pool Tomcatu. V novém se používá Jetty. Pokud používáte Tomcat, tak doporučuji používat novější dashboard (je tam opravená řada věcí, která v novější verzi Spring Bootu je malinko jinak) a předělat Jetty sekci tak, aby tam byly údaje z Tomcatu (je to opravdu hodně triviální).

Prometheus & batch jobs

- Prometheus se také může použít pro sběr metrik z batch jobů, v takovém případě se ale nepoužívá polling mechanismus, ale batch job provede push metrik do Pushgateway, odkud si to Prometheus získá.
 - <https://github.com/prometheus/pushgateway>

Prometheus databáze & PromQL

- Prometheus uchovává metriky ve své time series databázi:
 - <https://prometheus.io/docs/prometheus/latest/storage/>
 - Poznámka: Výchozí retence je 15 dnů, dá se samozřejmě zvýšit.
- Jako hodnoty (values) mohou být v databázi pouze čísla, proto například informace o tom, jestli server běží či nikoli (klíč “up”) nabývá hodnot 0 nebo 1.
- Pro dotazování do databáze slouží jazyk PromQL:
 - <https://prometheus.io/docs/prometheus/latest/querying/examples/>

Alertmanager

- Alertmanager je další server od tvůrců Prometheus, který umožňuje pomocí PromQL nastavit sadu pravidel (rules) a jakmile dojde k překročení nějakého pravidla, tak se pošle email:
 - <https://tomgregory.com/monitoring-a-spring-boot-application-part-3-rules-and-alerting/>
- Pěkný rozbor jak se počítají delays u alertů:
 - <https://pracucci.com/prometheus-understanding-the-delays-on-alerting.html>

Micrometer Extras

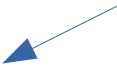
- Jednoduchá knihovna, která zobrazuje jak hodně paměti aplikace reálně používá (funguje jenom na Linuxu a Hotspotu):
 - <https://github.com/mweirauch/micrometer-jvm-extras>
- Má význam v situacích, kdy není možné použít Node exporter pro zjištění reálného použití paměti v systému (například u PaaS).

Zabezpečení Actuator endpointu

- Jak Prometheus provádí polling na actuator/prometheus endpoint, tak pokud endpoint není stateless, tak se při každém requestu vytvoří nová session. Jak tomu zabránit?

```
@Order(1)
@Configuration
public static class ActuatorWebSecurityConfigurerAdapter extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .antMatcher("/actuator/**")
            .authorizeRequests()
            .anyRequest()
            .hasRole("USER")
            .and()
            .httpBasic()
            .and()
            .sessionManagement()
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS);
    }
}
```

!!! Díky tomuto nastavení bude endpoint /actuator/** stateless!!!



Prometheus exporters & integrations

- Prometheus se v posledních letech postupně stal defacto standardem pro sběr metrik (i když se dá setkat se vším možným jako Splunk, ELK, Zabbix a další). Proto není divu, že hromada serverů a knihoven obsahuje možnost exportu dat v Prometheus formátu.
- Z Java knihoven to jsou například OkHttp, RestTemplate, Logback, Log4j2, Hibernate, ResilienceJ4 nebo různé knihovny / servery pro cachování jako EhCache nebo Hazelcast.
- Další servery a integrace jsou zde:
 - <https://prometheus.io/docs/instrumenting/exporters/>

Děkuji za pozornost