

LOG!



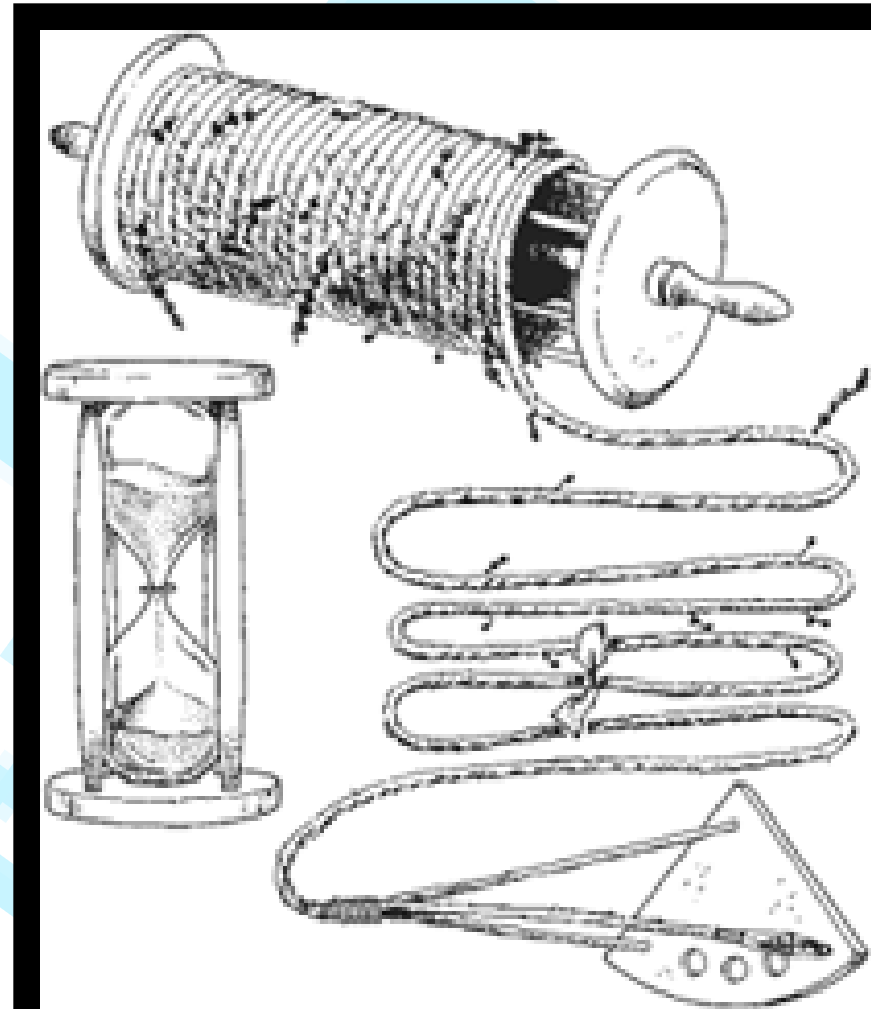
UINF/PAZ1c
epizóda 10

A classical painting of Prometheus, a Titan from Greek mythology. He is depicted as a muscular man with a beard and curly hair, wearing a red cape. He is holding a torch in his right hand and a rock in his left hand. He is looking up at the sky with a determined expression.

logovanie
metriky,
Prométeus

Logovanie

- Čo sa deje v systéme?
- Sledovanie medzikrokov algoritmu
- Čo sa stalo predtým, keď systém spadol?



System.out a System.err

- Štandardný a chybový výstup procesu
- Výpis iba na konzolu

```
Vyrátať som prvý krok  
x = 2.6621687e02  
Teraz rátam SELECT * FROM user;  
Vyrábam používateľa s id=1  
Vyrábam používateľa s id=2  
Vyrábam používateľa s id=3  
Neviem otvoriť súbor users.txt
```

Logovanie

System.out.println

java.util.logging

log4j

logback

slf4j



Dotiahnutie závislostí do pom.xml

```
<dependency>  
  <groupId>ch.qos.logback</groupId>  
  <artifactId>logback-classic</artifactId>  
  <version>VERZIA</version>  
  <scope>runtime</scope>  
</dependency>
```

Logback je závislý od slf4j takže maven nám dotiahne aj slf4j-api-x.x.x.jar

slf4j - použitie

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Arrays {
    private static final Logger logger
        = LoggerFactory.getLogger(Arrays.class);
}
```

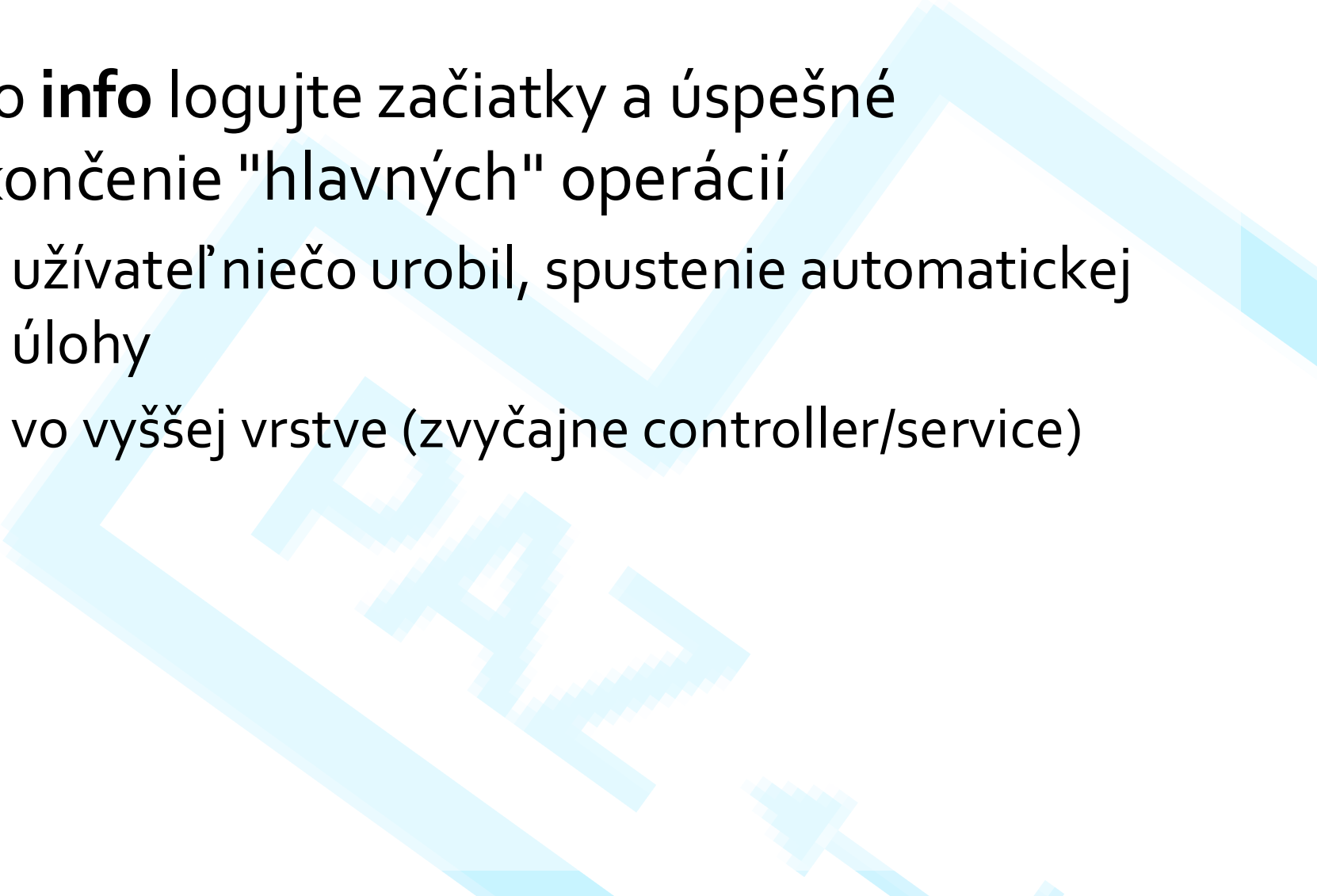
Úroveň podrobnosti logov

- fatal
- error
- warning
- **info**
- debug
- trace

```
try {  
    ...  
} catch (PripojenieKDatabaseException e) {  
    logger.error("Databaza nie je dostupna.", e);  
}
```

Čo logovať



- Do **info** logujte začiatky a úspešné skončenie "hlavných" operácií
 - užívateľ niečo urobil, spustenie automatickej úlohy
 - vo vyššej vrstve (zvyčajne controller/service)
- 

Čo logovať

- Logujte neúspešné konce "hlavných" udalostí
 - Odchytené výnimky
 - Zlyhaná validácia (niekde sme očakávali *true*, ale dostali sme *false*)
 - **error**; ak to neviete/nechcete riešiť (problém s DB spojením)
 - **warning**; ak to viete/chcete riešiť (súbor neexistuje - použijem default hodnoty; problém s DB spojením - počkám 5s a skúsim znova, alebo skúsim sekundárny DB server)
 - **info**; ak nastáva pomerne často (užívateľ nenájdenny, nesprávne heslo)
 - **error** v čo najvyššej vrstve (controller/service), **warning/info** čo najbližšie k zdroju výnimky

Čo logovať

- Do **debug** logujte volania komplexnejších metód
 - Volanie DB/vzdialeného API, otvorenie/zápis súboru, spustenie iného skriptu/programu, volanie funkcie z C knižnice, ...
 - Situačne: komplexnejšie výpočty (vyhľadávanie v grafe, transformácia dátových štruktúr)
 - Typicky, ale nie nutne, čo najbližšie k miestu volania
- **Trace** používajte iba situačne
 - Napr. chyba nastáva iba na vzdialenom serveri, kde nemáte debugger
 - Dočasne dodáte logger.trace pre každý riadok kódu v problémovej metóde a spustíte tam upravenú verziu

Aká je úroveň logovania – logback.xml

- vytvoríme súbor v src/main/resources

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%d{"yyyy-MM-dd'T'HH:mm:ss,SSSXXX", UTC} [%thread] %-5level
%logger{5} -- %msg -- %kvp%n</pattern>
    </encoder>
  </appender>

  <root level="INFO">
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```

alebo TRACE, DEBUG, INFO, ERROR, OFF

Logovanie v triede

- Ak chceme pre konkrétnu triedu logovať s iným levelom, rozšíríme logback.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%d{"yyyy-MM-dd'T'HH:mm:ss,SSSXXX", UTC} [%thread] %-5level %logger{5}
-- %msg -- %kvp%n</pattern>
    </encoder>
  </appender>

  <root level="WARN">
    <appender-ref ref="STDOUT" />
  </root>
  <logger name="sk.ics.upjs.entrance.UserEditController" level="TRACE" />
</configuration>
```

Logovanie s parametrami

- Klasický spôsob

```
logger.info("Validating user access with chip id={}", chipId,  
cardReaderId);
```

```
... INFO s.u.i.p.EntranceServiceImpl - Validating user with chip id=47656 to a card reader with id=1701
```

- Štruktúrované logovanie
 - Ľahšie sa parsuje a strojovo spracováva
 - Možno priamo vypisovať ako JSON, logfmt

```
logger.atInfo().  
    .addKeyValue("op", "validateAccess")  
    .addKeyValue("chipId", chipId)  
    .addKeyValue("cardReaderId", cardReader.getId())  
    .log("Validating user access to card reader");
```

```
... INFO s.u.i.p.EntranceServiceImpl -- Validating user access to a card reader --  
op="validateUserAccess" chipId="47656" cardReaderId="1701"
```

Čo má log obsahovať

- **Presný čas**,
 - RFC3339/ISO8601 formát
- **Typ logu** (INFO, ERROR, ...)
- **Identifikátor operácie**
 - Operácia: user klikne na tlačidlo, niekto zavolá sieťové API, atď.
 - typický ID vlákna pre väčšinu Java aplikácií*
 - vlastné umelé ID ak používame aplikačný framework/knižnicu na báze korutín
 - TRACE_ID pri distribuovaných systémoch (mikroslužby)
 - Identifikátor zdieľame pre všetky logy z operácie
- Približné miesto logu (trieda, názov zdrojového súboru, ...)
- **Stručná správa čo sa robilo** (načítaváme/vyrábame niečo)

```
2023-10-29T17:56:40,827Z INFO [thread-123] s.u.i.p.EntranceServiceImpl -- Validating user
– op="validateUserAccess" chipId="47656" cardReaderId="1701"
```

Čo má log obsahovať

- ...

- **Parametre**

- Typický ID entity/entít
- Akú operáciu sme robili – napr. názov metódy
- NIKDY nelogujte mená osôb, emaily, heslá, ani iné osobné údaje

```
logger.atInfo().  
    .addKeyValue("op", "validateUserAccess")  
    .addKeyValue("chipId", chipId)  
    .addKeyValue("cardReaderId", cardReader.getId())  
    .log("Validating user");
```

```
2023-10-29T17:56:40,827Z INFO [thread-123] s.u.i.p.EntranceServiceImpl -- Validating user  
– op="validateUserAccess" chipId="47656" cardReaderId="1701"
```

Koľko logovať

- Správne logovanie je umenie
- Príliš málo logov
 - Neviete čo sa presne stalo v prípade problému
- Príliš veľa logov
 - Aplikácia papká veľa CPU, RAM, disku, siete
 - Pomalá aplikácia
 - Rýchlo vybíja baterku (mobilné appky)
 - Nestíhate logy agregovať (ELK, Loki, Splunk)
 - ...



Logujeme kde-kade, kde-čo, rôzne

- kam posielat' logy
 - na konzolu (**ConsoleAppender**)
 - do lokálnych súborov (**FileAppender**, **RollingFileAppender**)
 - do tabuľky v databáze (DBAppender)
 - na vzdialený server cez ssh (SSLSocketAppender)
 - na mail (SMTPAppender)
 - do systémových logov (SyslogAppender)
 - logovanie pre každého klienta zvlášť (SiftingAppender)
- rôzny level logovania pre rôzne balíčky
- rôzny level do rôznych appenderov
- a mnoho ďalšieho na <http://logback.qos.ch/manual/>

Práca s logmi

- Jedna vec je logy generovať
- Čo s nimi potom?
- Agregáčné nástroje
 - Zber logov, efektívne vyhľadávanie cez GUI aj API
 - Automatické notifikácie
 - ELK (ElasticSearch, Logstash, Kibana)
 - Najpoužívanéjšie, flexibílné, ťažkopádne
 - Grafana Loki (resp. OSS fork Vali)
 - Modernejšie riešenie, menšie nároky na HW
 - Prudký nárast na popularite
 - Splunk
 - Proprietárne



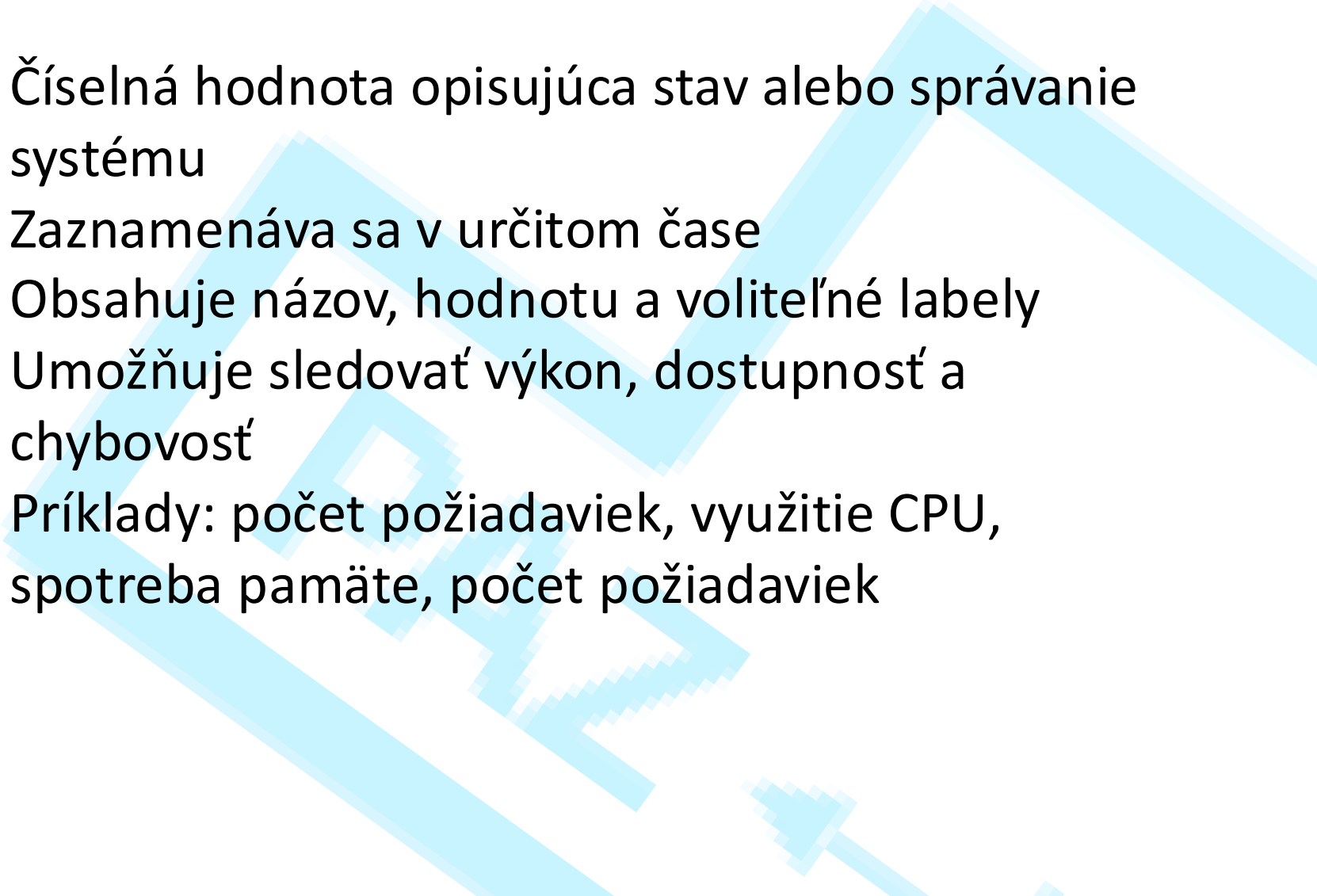
Nad rámec logovania



OpenTelemetry

- OTEL – Open Telemetry
 - Nový univerzálny štandard
 - Nástroje pre vš. jazyky a frameworky
- OTEL **logovanie**
 - Všeobecné rady a odporúčania ako používať existujúce log knižnice a nástroje
- **OTEL metriky**
 - Koľko CPU a RAM berú operácie? Ako dlho trvajú? Koľko dopytov robí každá operácia? Koľko máme chýb?
 - **Prometheus** (DB + API + GUI) + Grafana (GUI)
- OTEL **trasovanie (tracing)**
 - "Logovanie" pre distribuované systémy
 - Umožní spárovať logy z rôznych appiek
 - Jaeger (API + GUI)
 - Treba mu dodať DB (Cassandra, ElasticSearch)

Metrika



- Číselná hodnota opisujúca stav alebo správanie systému
- Zaznamenáva sa v určitom čase
- Obsahuje názov, hodnotu a voliteľné labely
- Umožňuje sledovať výkon, dostupnosť a chybovosť
- Príklady: počet požiadaviek, využitie CPU, spotreba pamäte, počet požiadaviek

Príklady metrík

- Spring Boot Actuator môže pre Prometheus poskytovať napríklad:
- **http_server_requests_seconds_count** – počet HTTP požiadaviek
- **http_server_requests_seconds_sum** – celkový čas spracovania požiadaviek
- **jvm_memory_used_bytes** – využitá pamäť JVM
- **jvm_threads_live_threads** – počet aktívnych vlákien
- **process_cpu_usage** – využitie CPU procesom
- **jvm_gc_pause_seconds_count** – počet prestávok spôsobených garbage collectorom
- ...
- Metriky sú dostupné cez endpoint */actuator/prometheus*

Labels

- Predstavujú dodatočné vlastnosti metriky
- Umožňujú filtrovanie a zoskupovanie dát
- Príklady: HTTP metóda, stavový kód alebo endpoint
- Každá kombinácia labelov vytvára nový časový rad
 - Príliš veľa rôznych hodnôt spôsobuje vysokú kardinalitu
 - Vysoká kardinalita - problémy s výkonom
- ID používateľov alebo požiadaviek **nie sú** vhodné ako labely
 - Metriky nenahrádzajú logy

Prometheus

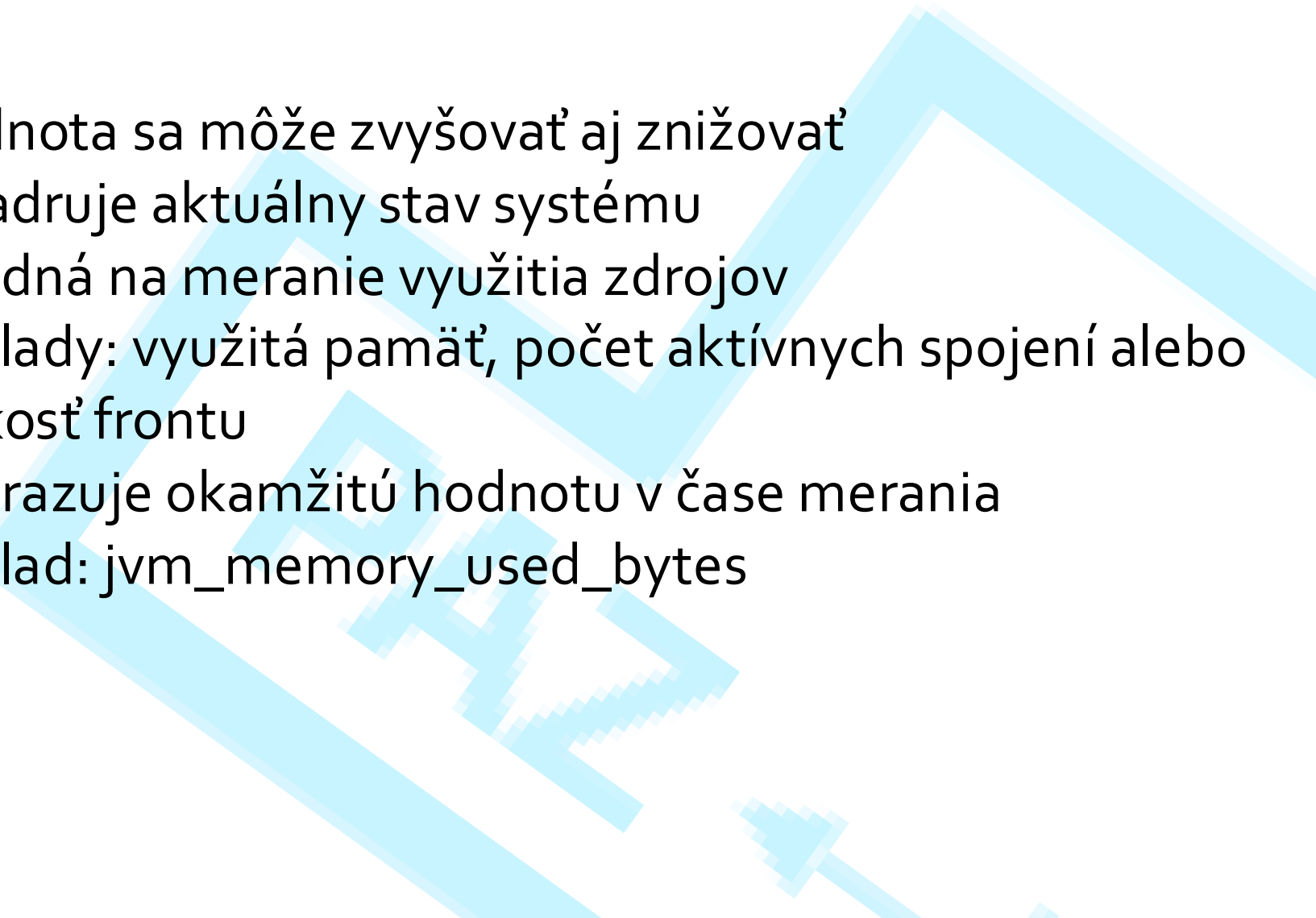


- Open-source systém na monitoring a alertovanie
- Pravidelne získava metriky z aplikácií pomocou HTTP
- Ukladá ich ako časové rady
- Metriky rozlišuje pomocou labelov
- Na dotazovanie používa jazyk **PromQL**
- Podporuje alerty cez **Alertmanager**
- Často sa vizualizuje pomocou **Grafany**
 - Má vlastné UI no je pomerne spartanské
 - Grafana podporuje vizualizáciu akýchkoľvek dát z akejkoľvek databázy

Counter metrika

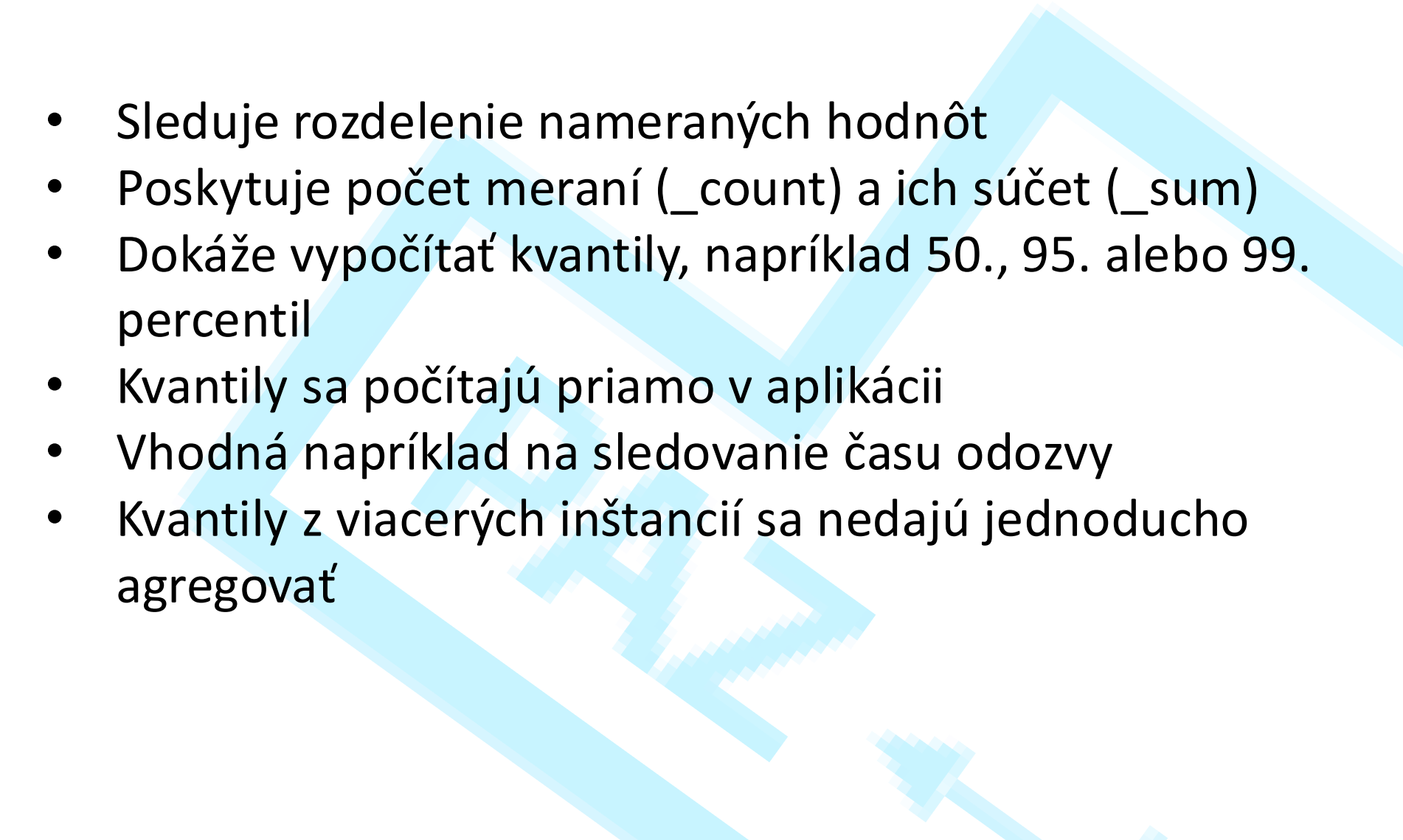
- Hodnota, ktorá sa iba zvyšuje
- Po reštarte aplikácie sa zvyčajne vynuluje
- Vhodná na počítanie udalostí
- Príklady: počet požiadaviek, chýb alebo spracovaných správ
- Často sa používa s funkciami `rate()` alebo `increase()`
- Príklad:
`http_server_requests_seconds_count`

Gauge metrika



- Hodnota sa môže zvyšovať aj znižovať
- Vyjadruje aktuálny stav systému
- Vhodná na meranie využitia zdrojov
- Príklady: využitá pamäť, počet aktívnych spojení alebo veľkosť frontu
- Zobrazuje okamžitú hodnotu v čase merania
- Príklad: `jvm_memory_used_bytes`

Summary metrika



- Sleduje rozdelenie nameraných hodnôt
- Poskytuje počet meraní (`_count`) a ich súčet (`_sum`)
- Dokáže vypočítať kvantily, napríklad 50., 95. alebo 99. percentil
- Kvantily sa počítajú priamo v aplikácii
- Vhodná napríklad na sledovanie času odozvy
- Kvantily z viacerých inštancií sa nedajú jednoducho agregovať

Histogram metrika

- Rozdeľuje namerané hodnoty do intervalov – bucketov
- Poskytuje hodnoty `_bucket`, `_count` a `_sum`
- Umožňuje počítat percentily pomocou PromQL
- Dáta z viacerých inštancií možno agregovať
- Presnosť závisí od zvolených hraníc bucketov
- Vhodná na meranie latencie alebo veľkosti požiadaviek

Prometheus v Dockri

```
services:
...
  prometheus:
    image: prom/prometheus:v3.14.0
    ports:
      - "9090:9090"
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml:ro
      - ./alerts.yml:/etc/prometheus/alerts.yml:ro
    extra_hosts:
      # Trik ako povoliť containeru pristupovať na hosta
      - "host.docker.internal:host-gateway"
```

prometheus.yml

```
global:
  scrape_interval: 15s # Ako často zbiera
  evaluation_interval: 15s # Ako často počíta alerty

rule_files:
  - /etc/prometheus/alerts.yml

scrape_configs:
  - job_name: entrance
    metrics_path: /actuator/prometheus
    static_configs:
      - targets: ["host.docker.internal:8080"]
```

Prometheus + Spring Boot

@Bean

```
SecurityFilterChain entranceSecurityFilterChain(  
    HttpSecurity http,  
    AuthenticationProvider entranceAuthenticationProvider)  
throws Exception {  
  
    http. ...  
        .authorizeHttpRequests(authorize -> authorize  
            .requestMatchers(HttpMethod.GET, "/actuator/prometheus").permitAll()  
            .anyRequest().authenticated()  
        )..  
}
```

Prometheus + Spring Boot

- Spring podporuje metriky

src/main/resources/application.properties

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

- Poskytne mnoho štandardných metrík pre Javu a obecné HTTP služby

Povolíme endpoint s metrikami (reálne by mal byť chránený s napr. mTLS)

SecurityConfig.java

@Bean

```
SecurityFilterChain entranceSecurityFilterChain(  
    HttpSecurity http,  
    AuthenticationProvider entranceAuthenticationProvider)  
throws Exception {  
  
    http. ...  
        .authorizeHttpRequests(authorize -> authorize  
            .requestMatchers(HttpMethod.GET, "/actuator/prometheus").permitAll()  
            .anyRequest().authenticated()  
        )..  
}
```

Alerty

- Upozorňujú na problémy zistené z metrík
- Podmienky sa definujú pomocou PromQL
- Prometheus pravidelne vyhodnocuje alertovacie pravidlá
- Parameter for zabraňuje upozorneniam na krátkodobé výkyvy
- Alertmanager zoskupuje a distribuuje upozornenia
- Notifikácie možno posielať e-mailom, do Slacku, sms, telefón
- Alerty môžu mať stavy **inactive**, **pending** a **firing**

Alerty

- **inactive**
 - podmienka alertu momentálne neplatí.
- **pending**
 - podmienka už platí, ale ešte neuplynul čas nastavený pomocou for.
- **firing**
 - podmienka platila počas celého času for; alert je aktívny a môže byť odoslaný do Alertmanagera.

Alerty

```
groups:
  - name: entrance
    rules:
      - alert: UnauthenticatedRequestsSurge

expr: increase(http_server_requests_seconds_count{status="401"}[1m]) >=
2

  for: 0s # 5m
  labels:
    severity: warning
  annotations:
    summary: "2+ pokusov o pristup bez prihlasenia"
    description: "401 request counter sa zvyšil o {{ $value }} v
jednej minute. Niektó sa tu chce naburat."
```

Vlastná metrika v RestController

```
@Autowired
private MeterRegistry meterRegistry;

private void recordFindById(Tags tags) {
    Timer.builder("entrance_users_find_by_id")
        .description("User lookup by id")
        .tags(tags)
        .register(meterRegistry)
        .record(() -> {});
}
```

Vlastná metrika v RestController

```
@GetMapping("/users/id/{id}")
public User findById(@PathVariable Long id) throws
    NotFoundException {
    try {
        var user = this.userDao.findById(id);
        recordFindById(Tags.of("outcome", "success"));
        return user;
    } catch (NotFoundException e) {
        recordFindById(Tags.of("outcome", "not_found"));
        throw e;
    }
}
```

Otázky?

