Performance tuning

Load testing

- Soap UI (pro základní scénáře stačí open-source edice, ale pro pokročilejší scénáře je zapotřebí placená verze)
  - https://www.soapui.org/load-testing/creating-and-running-loadtests.html
- Apache JMeter
  - https://jmeter.apache.org/
  - https://www.seleniumeasy.com/jmeter-tutorials/http-request-sampler-example
- Gatling
  - https://gatling.io/

Profiling

- Zdarma v JDK
  - JMC (Java Mission Control) + FlightRecorder – v OracleJDK zdarma pouze na dev & test, v OpenJDK zdarma i na produkci + Idea má integraci s JMC
    - https://openjdk.java.net/projects/jmc/7/
    - https://jdk.java.net/jmc/
    - https://github.com/JDKMissionControl/jmc
  - Visual VM
    - https://visualvm.github.io/
  - JConsole
- Placené
  - JProfiler
    - https://www.ej-technologies.com/products/jprofiler/overview.html
  - XRebel
    - https://zeroturnaround.com/software/xrebel/
  - Yourkit
    - https://yourkit.com/

JMC & Flight Recorder

- Použití Flight Recorder:
  - V Oracle JDK <= 8 se odblokuje při startu aplikace:

    ```
    -XX:+UnlockCommercialFeatures -XX:+FlightRecorder
    ```

  - Je zdarma všude kromě produkčního prostředí a pouze v Oracle JDK.
  - V OpenJDK >= 8 je zdarma, jenom v současnosti se musí provést build JMC. Při startu aplikace není zapotřebí žádné odblokování.
- Poté je možné ho v JMC zapnout.
  - NEBO:
  - Můžete Flight Recorder zapnout při startu aplikace, kde se dá také specifikovat kam se budou jeho soubory ukládat (ve výchozím nastavení do `/tmp`) a také se dá jeho start odložit:
    - https://docs.oracle.com/javacomponents/jmc-5-4/jfr-runtime-guide/run.htm#JFRUH164

Metrics

- Prometheus
  - https://prometheus.io/
- NewRelic (jenom v Cloudu, placené)
  - https://newrelic.com/
- Java Melody
  - https://github.com/javamelody/javamelody/wiki

Actuator

- Spring Boot Actuator

```
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details=always
```

- Spring Boot Admin:
  - https://github.com/codecentric/spring-boot-admin
    - Řada věcí není aktivní „out-of-the-box“, musí se aktivovat.

Analýza Heapu

- Eclipse MAT, visualvm, jprofiler

Analýza logů

- Kibana nebo Graylog (nebo Splunk):
  - https://www.graylog.org/
- https://goaccess.io/

- Přidávat MDC (Mapped Diagnostic Context)!!!

Tracing

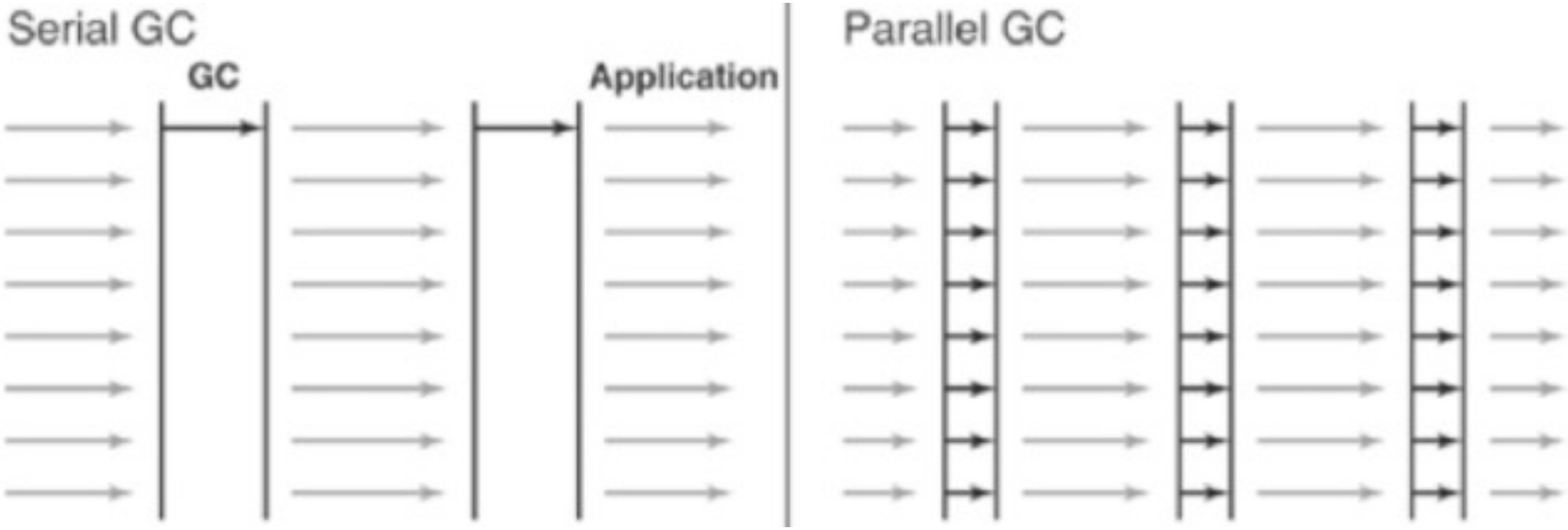
- Tracing = odhalování příčin latence požadavků u microservice architektury
- Nástroje:
  - Zipkin:
    - https://github.com/openzipkin/zipkin
  - Jaeger:
    - https://www.jaegertracing.io/
  - Spring Boot Admin má jenom základní přehled

APM

- Kibana APM
- Glowroot
  - https://glowroot.org/
- AppDynamics (placené)
  - https://www.appdynamics.com/

Garbage Collector I.

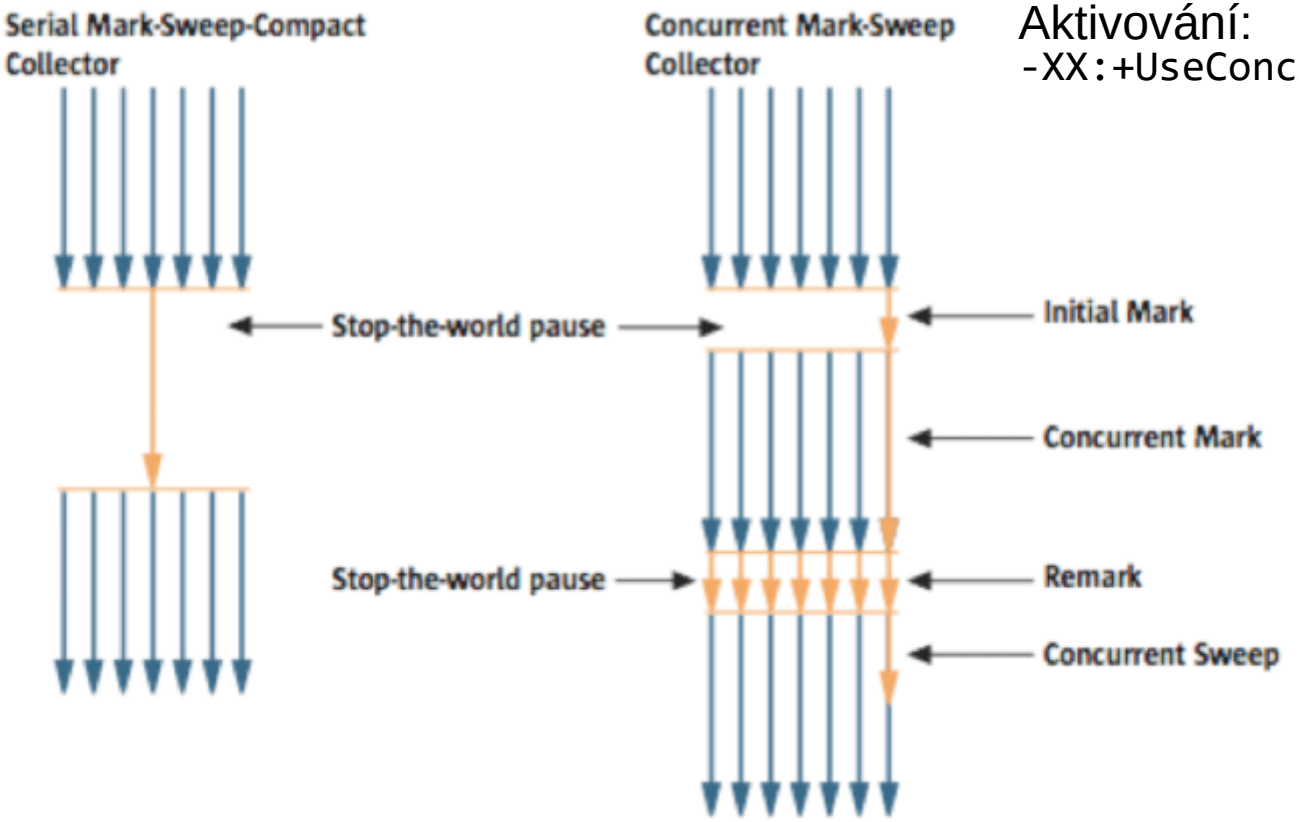
- Existuje několik typů Garbage Collectorů (GC):
  - Serial GC: Běží v jednom vlákně, ale při svém běhu zastaví celou aplikaci (Stop The World - STW).
  - Parallel GC: Běží ve více vláknech, ale také zastaví celou aplikaci. Do Java 8 včetně výchozí GC.



Garbage Collector II.

- (Concurrent Mark Sweep) CMS GC: Běží ve více vláknech a zastaví aplikaci v počáteční fázi kdy si označí objekty v paměti ke smazání, poté čistí paměť zatímco vlákna aplikace běží a další STW provede po ukončení běhu GC. Od Java 11 deprecated.

Serial GC vs. CMS:



Aktivování:
`-XX:+UseConcMarkSweepGC`

Garbage Collector III.

- Nevýhoda CMS GC je, že spotřebuje více CPU než předcházející GC. Výhoda je, že STW je výrazně kratší.
- Nejnovější a od Java 9 výchozí GC je G1 (od Java 7u4):
  - Používá více vláken a rozdělí paměť do částí o velikosti 1 až 32 MB (záleží na velikosti heapu) a potom se při čištění zaměří na části, které mají nejvíc neplatných objektů (G1 = Garbage First).
  - Od Java 8u20 má String deduplication – G1 hledá v paměti duplikované Stringy a odstraňuje kopie.
  - `-XX:+UseG1GC -XX:+UseStringDeduplication`
  - Pěkný článek na téma String deduplication:
    - https://blog.codecentric.de/en/2014/08/string-deduplication-new-feature-java-8-update-20-2/

Materiály

- http://blog.takipi.com/garbage-collectors-serial-vs-parallel-vs-cms-vs-the-g1-and-whats-new-in-java-8/
- http://www.cubrid.org/blog/dev-platform/understanding-java-garbage-collection/

Analýza paměti – příkazová řádka

- Paměť můžete analyzovat pomocí jakéhokoli profileru. Nicméně je možné analyzování paměti zapnout také z příkazové řádky pomocí:

```
-XX:-PrintGC -XX:-PrintGCDetails -Xloggc:gc.log
```

- Poté se logy uloží do souboru gc.log. Pro jeho analýzu je moc pěkný web:
  - http://gceasy.io/
  - Poznámka: Od stejného autora je také tento web pro analýzu threadů:
    - https://fastthread.io/

Přepínače

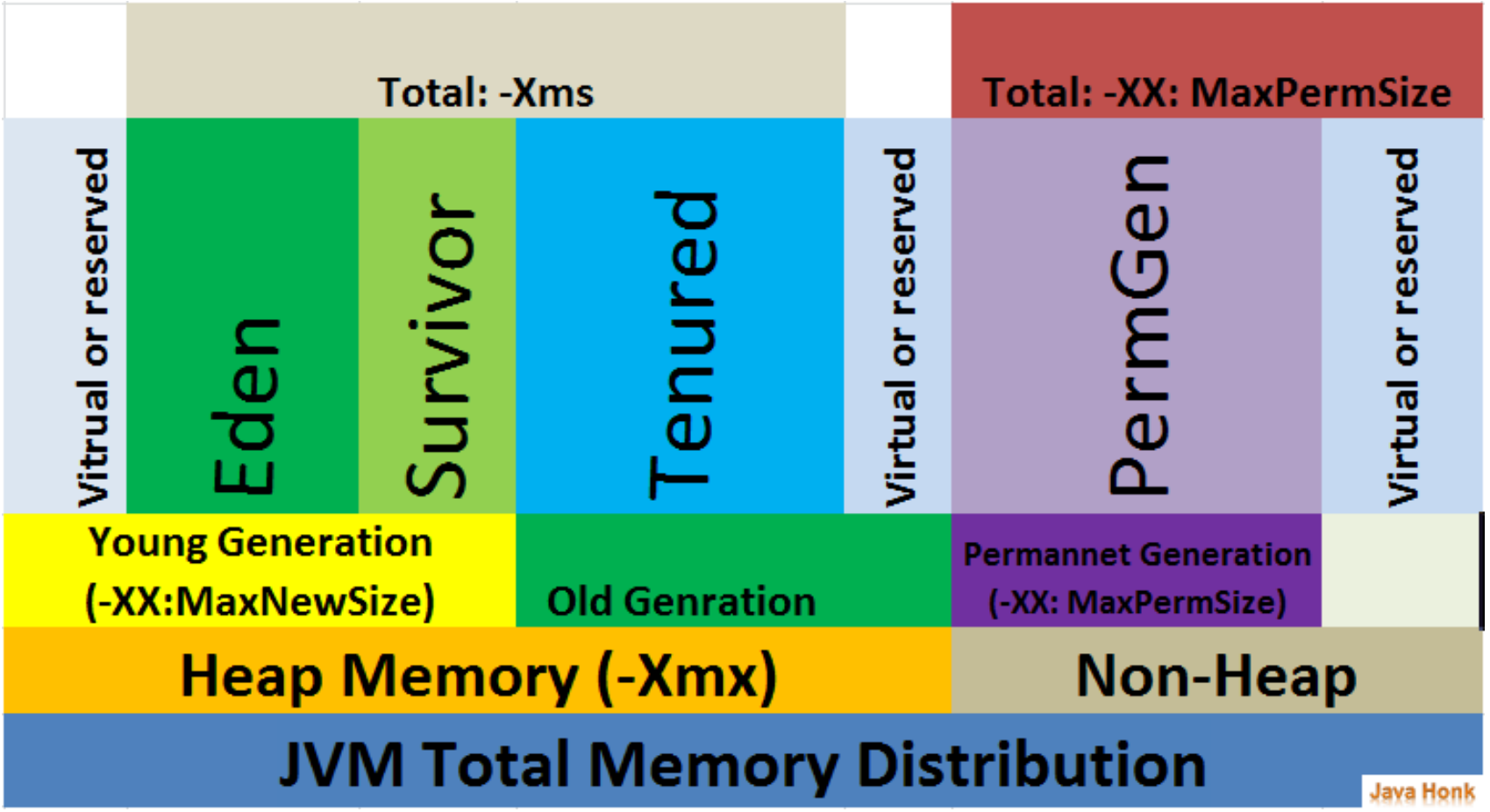
- Při startu aplikace se dá nastavit celá řada věcí:

  `-Xmx=1g` (maximální množství paměti)

  `-XX:+HeapDumpOnOutOfMemory` (když dojde k OutOfMemoryError, tak provede heap dump)

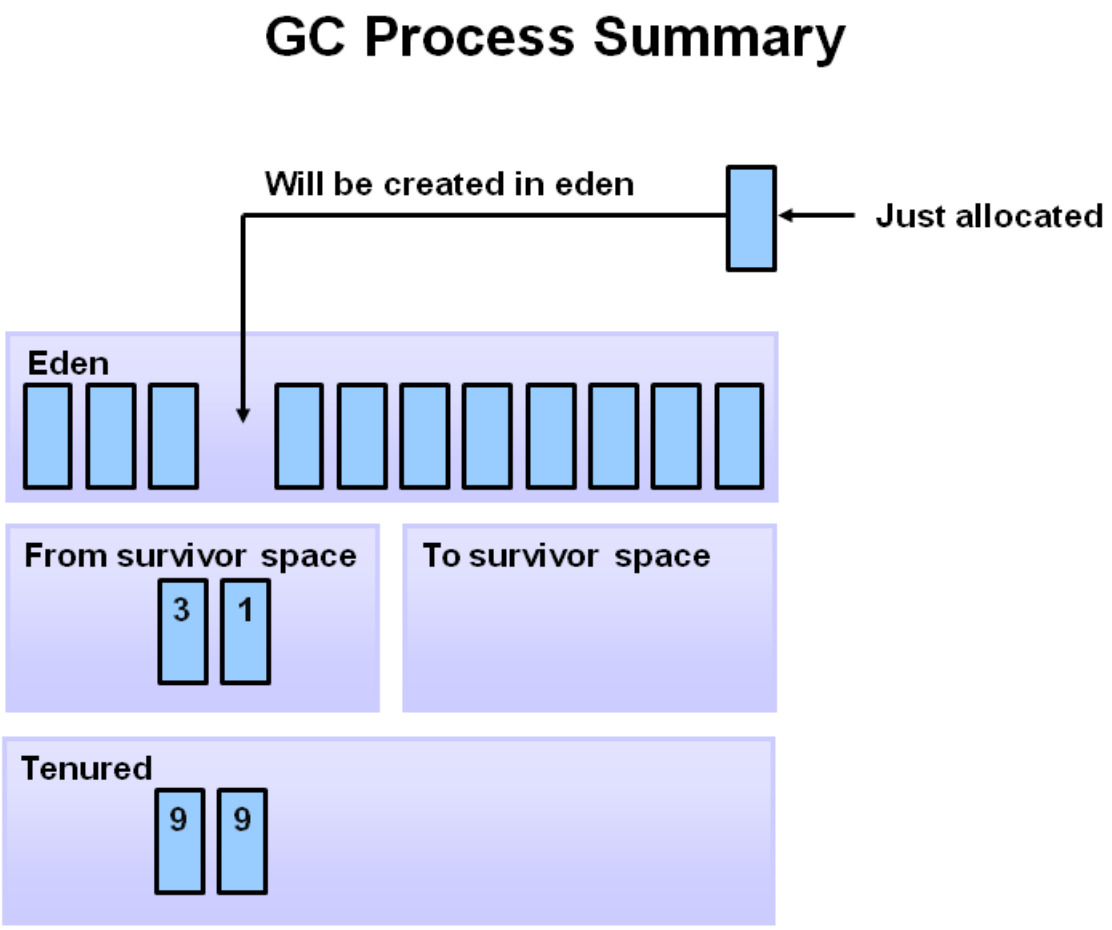
- Celý seznam:
  - http://docs.oracle.com/javase/8/docs/technotes/tools/windows/java.html

Typy pamětí I.



Poznámka: V Java 8 byl PermGen nahrazen Metaspace. Základní rozdíl je, že PermGen byl ve výchozím nastavení omezený, zatímco Metaspace je neomezený:
http://stackoverflow.com/questions/27131165/what-is-the-difference-between-permgen-and-metaspace

Typy pamětí II.



http://stackoverflow.com/questions/2129044/java-heap-terminology-young-old-and-permanent-generations

Tipy

- Logování je obvyklý bottleneck. V současnosti je nejlepší Logback nebo Log4J 2:
  - https://logging.apache.org/log4j/2.x/performance.html
- Co je ultra pomalé:
  - Výjimky
  - String.replace (do Java 8 včetně), String.split
  - Spojování Stringů pomocí plus operátoru v cyklu
  - Regulární výrazy
  - Reflexe
  - Dozer, Orika, nejlepší je MapStruct:
    - https://github.com/jirkapinkas/java-mapping-frameworks

Tipy

- Jackson AfterBurner:
  - https://github.com/FasterXML/jackson-docs/wiki/Presentation:-Jackson-Performance
  - https://github.com/FasterXML/jackson-modules-base/tree/master/afterburner
  - https://stackoverflow.com/questions/49454849/how-can-i-register-and-use-the-jackson-afterburnermodule-in-spring-boot
- Pozor na CORS na produkci:
  - https://www.freecodecamp.org/news/the-terrible-performance-cost-of-cors-api-on-the-single-page-application-spa-6fcf71e50147/
- Hikari CP:
  - https://github.com/brettwooldridge/HikariCP

Tipy

- Microbenchmarking správně:
  - Testování rychlosti nějaké metody není tak jednoduché jak by na první pohled mohlo vypadat, JVM má řadu optimalizací, díky kterým se kód může v různých prostředích chovat různým způsobem. Jak správně „tunit“ nějakou metodu?
    - http://tutorials.jenkov.com/java-performance/jmh.html
    - http://www.rationaljava.com/2015/02/jmh-how-to-setup-and-run-jmh-benchmark.html
    - https://www.researchgate.net/publication/333825812_What%27s_Wrong_With_My_Benchmark_Results_Studying_Bad_Practices_in_JMH_Benchmarks
      - Praktický příklad:
        - https://www.baeldung.com/java-performance-mapping-frameworks

Novinky v Java 8

- Streamy jsou o malinko pomalejší (1 – 2%), ale čitelnější než imperativní způsob programování.
  - Pozor na parallel stream – ve výchozím nastavení se používá pool vláken, ve kterém je (počet jader – 1) vláken. Toto se dá změnit (není best practice, best practice je nepoužívat parallel stream):
    - https://stackoverflow.com/questions/21163108/custom-thread-pool-in-java-8-parallel-stream
- Na pořadí operací ve streamu záleží, filtry zásadně na začátek.
- Nové Date-Time API je rychlejší než staré (i než Joda-time).
- Lambdy jsou stejně rychlé jako anonymní třídy (akceptovaná odpověď)
  - https://stackoverflow.com/questions/34585444/java-lambdas-20-times-slower-than-anonymous-classes

ZGC (od Java 11)

- Nový Garbage Collector (zatím v experimentální fázi) pro obrovské (multi-terabyte) heapy:
  - https://www.opsian.com/blog/javas-new-zgc-is-very-exciting/

Docker

- Buildit aplikaci do JAR souboru a ten přidávat do Docker image zbytečně plýtvá prostředky. Lepší je:
  - Buď použít JIB
    - https://github.com/GoogleContainerTools/jib
  - Nebo (pouze pro Spring Boot):
    - https://spring.io/blog/2020/08/14/creating-efficient-docker-images-with-spring-boot-2-3

Práce s databází

Nejvíc low-level (nejrychlejší)			Nejvíc high-level (nejpomalejší)		
JDBC	JdbcTemplate	MyBatis	Hibernate (Session)	Hibernate (EntityManager)	Spring Data JPA

JDBC Batch

- Typicky při importech dat je dobrý nápad používat JDBC Batch. Když provedete 1 000 000 x insert, tak když to budete volat jeden insert po druhém, tak za jednu vteřinu jich vykonáte pouze řádově desítky.
- Při použití JDBC Batch (ať už low-level JDBC, nebo JdbcTemplate) to budou desítky tisíc operací za vteřinu (tohle je hezky vidět v JProfileru).
- https://www.baeldung.com/jdbc-batch-processing

Hibernate

- Logovat SQL a řešit problémy IHNED (a dívat se jaký SELECT Hibernate vygeneruje … občas dokáže zbytečně vygenerovat i kartézský součin)!!!
- Mít v application.properties: spring.jpa.open-in-view=false
- Při používání vazeb může dojít k n+1 problému. Jak tomu předcházet:
  - Nepoužívat @OneToOne a @ManyToMany vazby pokud nevíte co děláte!!!
  - Používat LAZY vazby a v případě potřeby vygenerovat join pomocí „join fetch“ nebo @EntityGraph
    - Join fetch ale volat pouze když to dává smysl z pohledu SQL!!!
      - Volat join fetch na vazbě @ManyToOne je bezpečné vždy.
      - Tzn. obecně čím méně SELECTů do databáze, tím lépe. Ale ne vždy!!!

Transakce

- @Transactional nebo @Transactional(readOnly=true) generuje COMMIT do databáze. Je dobré přemýšlet při používání této anotace aby se zbytečně neposílalo velké množství COMMITů do databáze při čtení dat:
  - Když v service metodě zavoláme 3 findXXX metody z repozitáře, pak se 3x zavolá SELECT a 3x COMMIT! Když na tu metodu v service dáme @Transactional(readOnly=true), pak se vyvolají 3 SELECTy a jenom jeden COMMIT!

```
void doStuff() {
      repo.findAll();
      repo.findAll();
      repo.findAll();
}
```

3 transakce

vs.

```
@Transactional(readOnly = true)
void doStuff() {
      repo.findAll();
      repo.findAll();
      repo.findAll();
}
```

1 transakce

U Spring Data JPA: findXXX() operace jsou automaticky oanotované s @Transactional(readOnly = true), save(), delete ... operace jsou oanotované s @Transactional. @Query metody musíte oanotovat vy sami.

SQL Tips & Tricks

- Obecně použití indexu je lepší než sekvenční přístup (ale ne vždy).
- Každý INDEX prodlouží délku DML operací (INSERT, UPDATE, DELETE) trojnásobně!
- Ne vždy může databáze použít index, například zde se index použít nemůže:
  - select * from item where name like '%neco%'
    - Je to kvůli tomu, že je procento na začátku, kdyby bylo jenom na konci, tak se index zavolat může.
- Indexy & EXPLAIN PLAN … základ k pochopení proč jsou SELECTy pomalé!

SQL Tips & Tricks

- Vždy používat PreparedStatement!!!
- Nikdy neposílat do databaze:
  - select * from customer where id = 1 HARD PARSING
  - select * from customer where id = 2 HARD PARSING
  - select * from customer where id = 3 HARD PARSING

- Musí se posílat:
  - select * from customer where id = :id HARD PARSING
    - id = 1
  - select * from customer where id = :id SOFT PARSING
    - id = 2
  - select * from customer where id = :id SOFT PARSING
    - id = 3

Thread pool v Tomcatu

- Thread pool v Tomcatu ve výchozím nastavení obsahuje 200 vláken, které sdílí všichni klienti. To může být málo. Řešením je:
  - Buď optimalizovat požadavky klientů tak, aby byly rychleji odbavovány a tím nezatěžovali tolik tento pool vláken
  - Nebo řešit déle trvající operaci v jiném poolu vláken:
    - https://www.baeldung.com/spring-deferred-result
  - Nebo zvýšit hodnotu server.tomcat.max-threads:
    - https://stackoverflow.com/questions/25399400/maximum-client-request-thread-pool-size-in-spring
  - Nebo použít reaktivní způsob programování a místo Tomcatu použít Netty server.

urandom

- https://ruleoftech.com/2016/avoiding-jvm-delays-caused-by-random-number-generation

Awesome *

- https://github.com/akullpp/awesome-java/blob/master/README.md#performance-analysis
- https://github.com/akullpp/awesome-java/blob/master/README.md#monitoring
- https://github.com/mfornos/awesome-microservices/blob/master/README.md#monitoring-and-debugging