

# Microservices

# Výhody

- Méně komplexní a jednodušší na porozumění
- Redukce side efektů
- Lepší škálování
- Polyglot
- Týmy mohou pracovat nezávisle (když dodrží kontrakt mezi službami)
- Release cycle jednotlivých služeb je na sobě nezávislý

# Bounded Context

- Jak definovat scope microservicy? To se definuje pomocí tzv. Bounded Context
  - <https://martinfowler.com/bliki/BoundedContext.html>
  - Prakticky se rozdělí jeden velký model do menších Bounded Contextů a definují se vztahy mezi nimi
- Best practices:
  - 1 microservice = 1 bounded context
  - Snaha vyhnout se výměně dat mezi microservicemi (tomu se ale nejde úplně vyhnout)

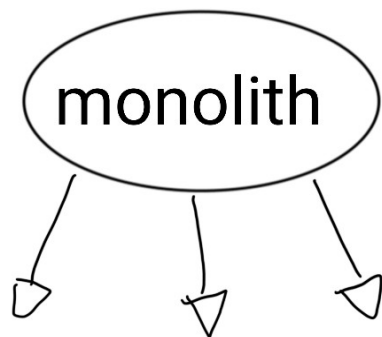
# Modelování microslužeb

- Microslužby:
  - Jsou definovány okolo business funkcionalit, nikoli horizontálních vrstev jako data access / messaging
  - Microservices should have loose coupling and high functional cohesion
    - Microservica je “loosely coupled”, když je možné změnit jednu microservicu bez nutnosti změny jiné microservicy
    - Microservica je “cohesive”, když má jeden účel (nedělá X věcí)
    - <https://docs.microsoft.com/en-us/azure/architecture/microservices/model/domain-analysis>
  - Microslužby by měly splňovat (nebo vycházet) z 12 factor app principů:
    - <https://12factor.net/>

# Datová vrstva: základní pravidla

- Každá microservice má svoji databázi
- Microservisy nepracují s databázemi jiných microservis
  - Samozřejmě to technicky možné je, ale sníží se tím škálovatelnost microservis:
    - <https://microservices.io/patterns/data/database-per-service.html>
    - <https://microservices.io/patterns/data/shared-database.html>
- Pouze lokální transakce (nikoli distribuované transakce)
- Bonus: Automatizovaná aktualizace databázového schématu (Liquibase, Flyway)

# Čím začít? Monolith, microservices nebo modulární monolith?



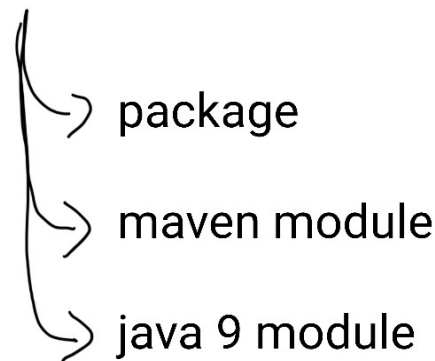
microservices

může být těžké  
modularizovat

microservices

může být overkill

modular  
monolith



zlatá střední cesta.  
Tvoří se monolith,  
ale lze ho lehce  
modularizovat

# Komunikace Microservices

- Obyčejně se používá REST (posílají se JSON soubory). Je ale možné použít cokoli (XML, ...), nebo také GraphQL:
  - <https://graphql.org/>
- Pro vygenerování jednoduchého API pro číselníky je skvělý Spring Data REST:
  - <https://spring.io/projects/spring-data-rest>

# Problém: sdílení dat I.

- Čtení:
  - Přímé volání služeb & API composer design pattern
    - <https://microservices.io/patterns/data/api-composition.html>
    - <https://learn.co/lessons/microservices-patterns-chapter-7>
    - Provedou se dotazy na jednotlivé microservicery, získají se výsledky a spojí se dohromady in-memory.



# Problém: sdílení dat II.

- Čtení:
  - Replikace dat
    - [https://medium.com/@john\\_freeman/querying-data-across-microservices-8d7a4667668a](https://medium.com/@john_freeman/querying-data-across-microservices-8d7a4667668a)
    - <https://medium.com/trendyol-tech/event-driven-microservice-architecture-91f80ceaa21e>
    - NEBO: Event Sourcing
      - Tohle je úplně jiný typ architektury
      - <https://microservices.io/patterns/data/event-sourcing.html>

# Problém: sdílení dat III.

- Konzistentní zápis dat
  - SAGA pattern
    - <https://microservices.io/patterns/data/saga.html>
    - Choreography nebo Orchestrator

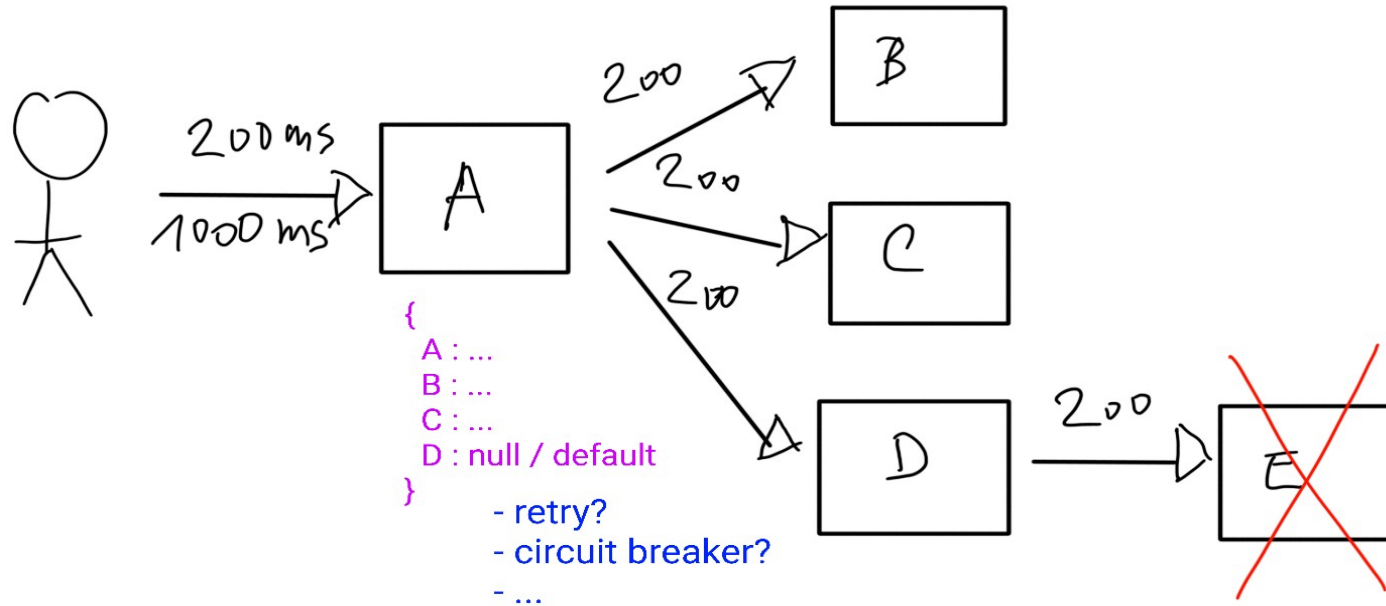
# Problém: sdílení dat III.

- Něco mezi (tohle není microservice architecture, ale pro některé typy aplikací se to může hodit)
  - Databázová VIEW
    - [https://medium.com/@john\\_freeman/querying-data-across-microservices-8d7a4667668a](https://medium.com/@john_freeman/querying-data-across-microservices-8d7a4667668a)
    - Každá microservice má zvlášť schéma ve stejné databázi. Pro zpřístupnění dat jiné microservice se použije DB VIEW.

# Circuit Breaker I.

- Problém:
  - V API Composer design patternu se napřímo volá X microservis. Co když je ale nějaká pomalejší s odpovědí, nebo dokonce vůbec nefunguje? V takové situaci se může hodit Circuit Breaker:
    - <https://github.com/resilience4j/resilience4j>
    - Dřív se používal Netflix Hystrix, postupně byl nahrazen Resilience4j
- Princip:
  - Circuit Breaker monitoruje requesty na ostatní služby a jakmile nějaká služba přestane odpovídat, vrátí nějaký definovaný fallback.

# Circuit Breaker / Retry



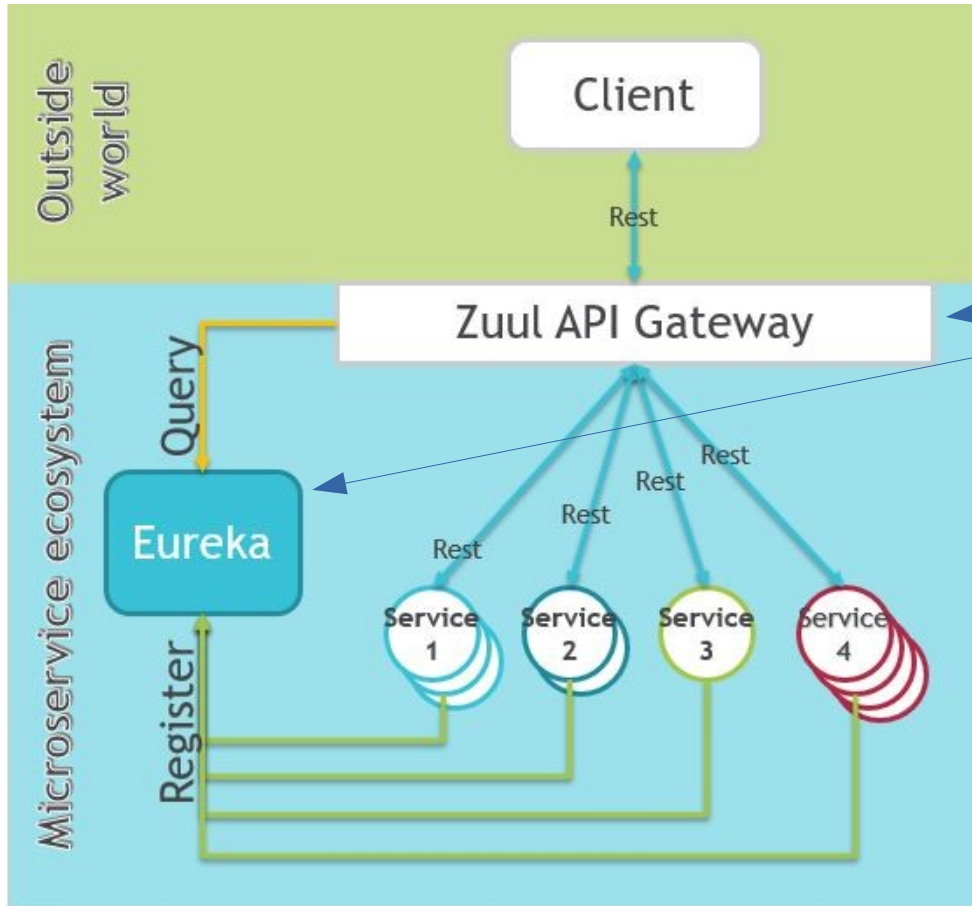
# Circuit Breaker II.

- Výhody použití Circuit Breaker:
  - Šetří lokální prostředky, které by bez Circuit Breakeru byly blokovány dokud by nedošlo například ke timeoutu nebo nějaké chybě
  - Je prevencí pro přetížení ostatních služeb
  - Díky němu se vyhneme kaskádovému efektu kdy jedna služba nefunguje a to ovlivňuje jiné služby
  - Umožňuje vrátit klientovi fallback když služba nefunguje / je pomalá
- <https://github.com/resilience4j/resilience4j-spring-boot2-demo>

# Retry, Cacheable

- Resilience4j také podporuje retry mechanismus a cachování. K tomu ale není zapotřebí tato knihovna, “stačí” Spring:
  - Spring Retry
    - <https://www.baeldung.com/spring-retry>
  - Spring Cacheable
    - <https://www.baeldung.com/spring-cache-tutorial>
    - <https://www.baeldung.com/spring-boot-ehcache>
    - <https://www.foreach.be/blog/spring-cache-annotations-some-tips-tricks>
    - <https://hazelcast.org/>

# Spring Cloud Netflix





# Service Registry

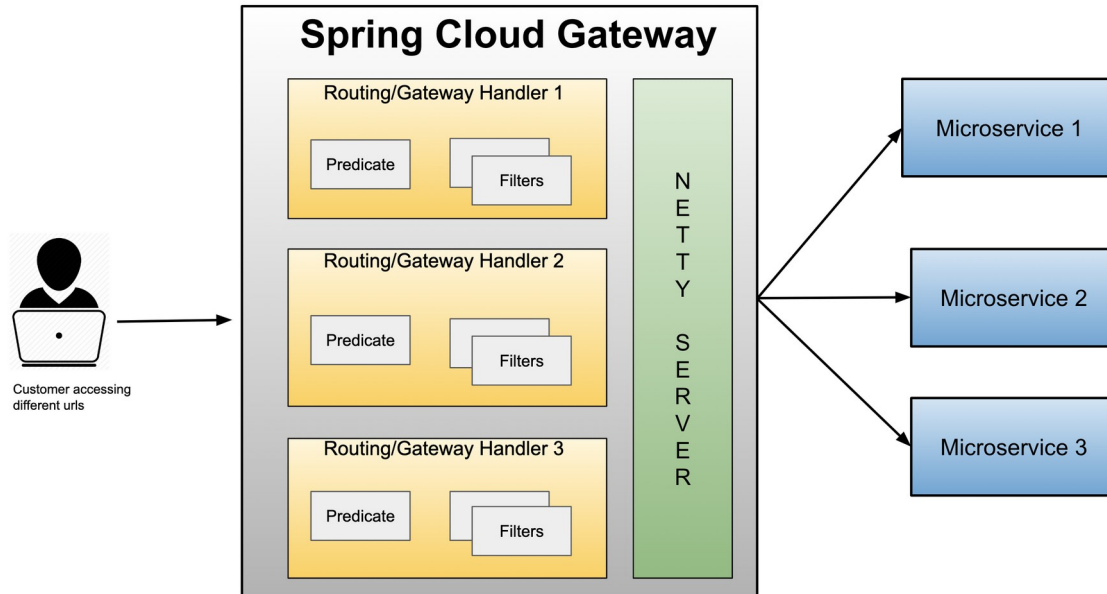
- Service registry = telefonní seznam pro microservic
- Když se service nashartuje, zaregistruje se pod textovým názvem v Service registry. Když chceme volat service, zeptáme se Service registry a získáme adresu service.
- Registry:
  - Netflix Eureka
  - Apache Zookeeper
  - Hashicorp Consul

# Open Feign

- Deklarativní REST klient
  - <https://spring.io/projects/spring-cloud-openfeign>
- Integruje se s:
  - Service Discovery (Eureka / Zookeeper)
    - Poznámka: je možné použít i bez Service Discovery!
  - Circuit Breaker (Hystrix / Resilience4j)
  - Load Balancer (Ribbon)
- Ribbon
  - Load Balancer

# Gateway (Zuul)

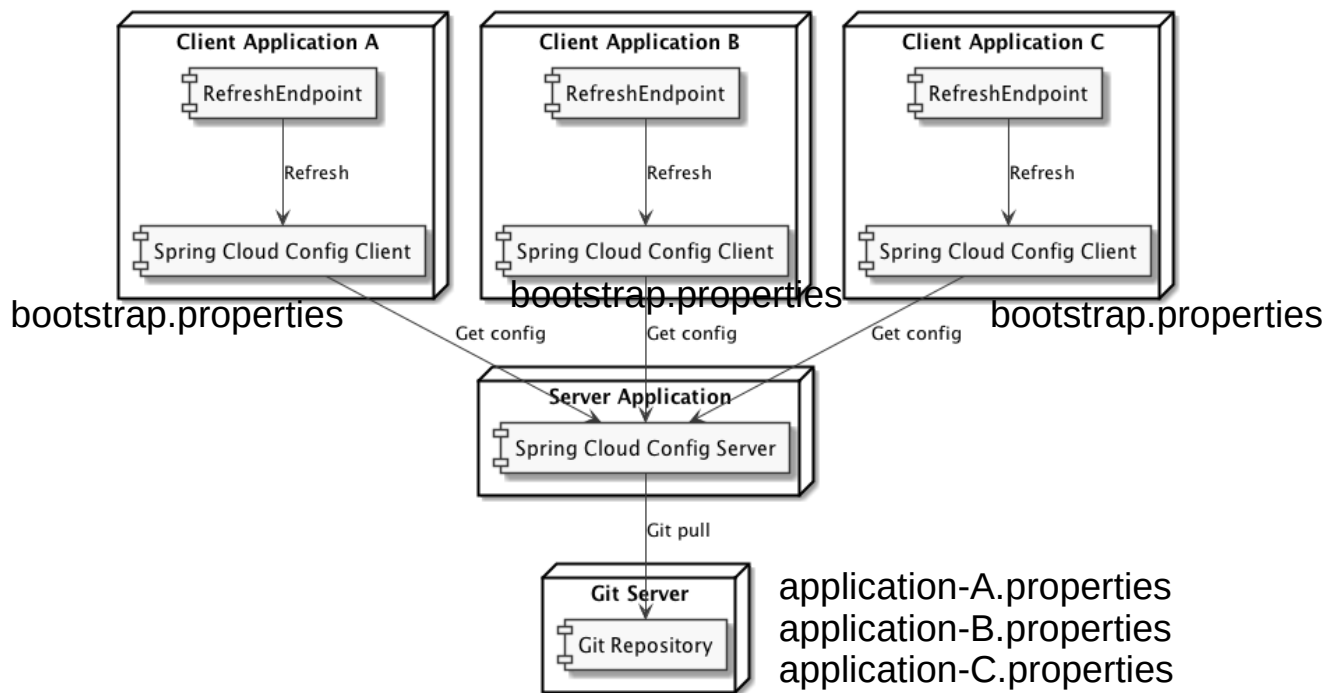
- Spring Cloud Gateway (dříve Zuul) je součástí Spring Cloud Netflix balíku mikroslužeb a slouží jako reverse-proxy server pro mikroslužby:
  - <https://medium.com/@niral22/spring-cloud-gateway-tutorial-5311ddd59816>



Poznámka: Bez Spring Cloud Netflix se používá Traefik:  
<https://github.com/traefik/traefik>

# Spring Cloud Config

- Spring Cloud Config slouží pro centralizované uchovávání konfiguračních souborů (application.properties) na jednom místě (v GITu):
  - <https://cloud.spring.io/spring-cloud-config/reference/html/>



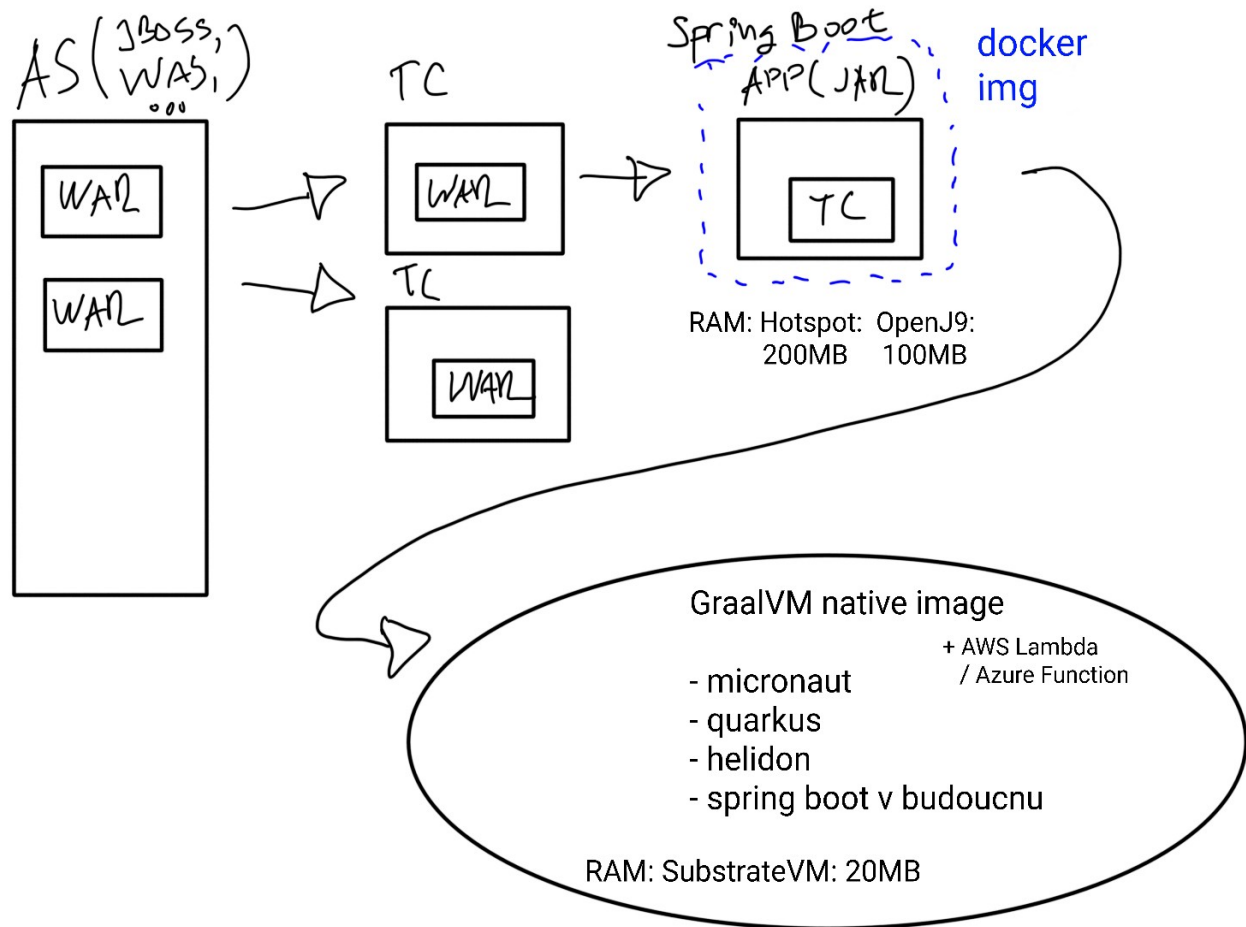
# Vault

- Vault se dá použít buď napřímo, nebo prostřednictvím Spring Cloud Config serveru:
  - <https://www.vaultproject.io/>
  - <https://spring.io/guides/gs/vault-config/>
  - <https://blog.marcosbarbero.com/integrating-vault-spring-cloud-config/>

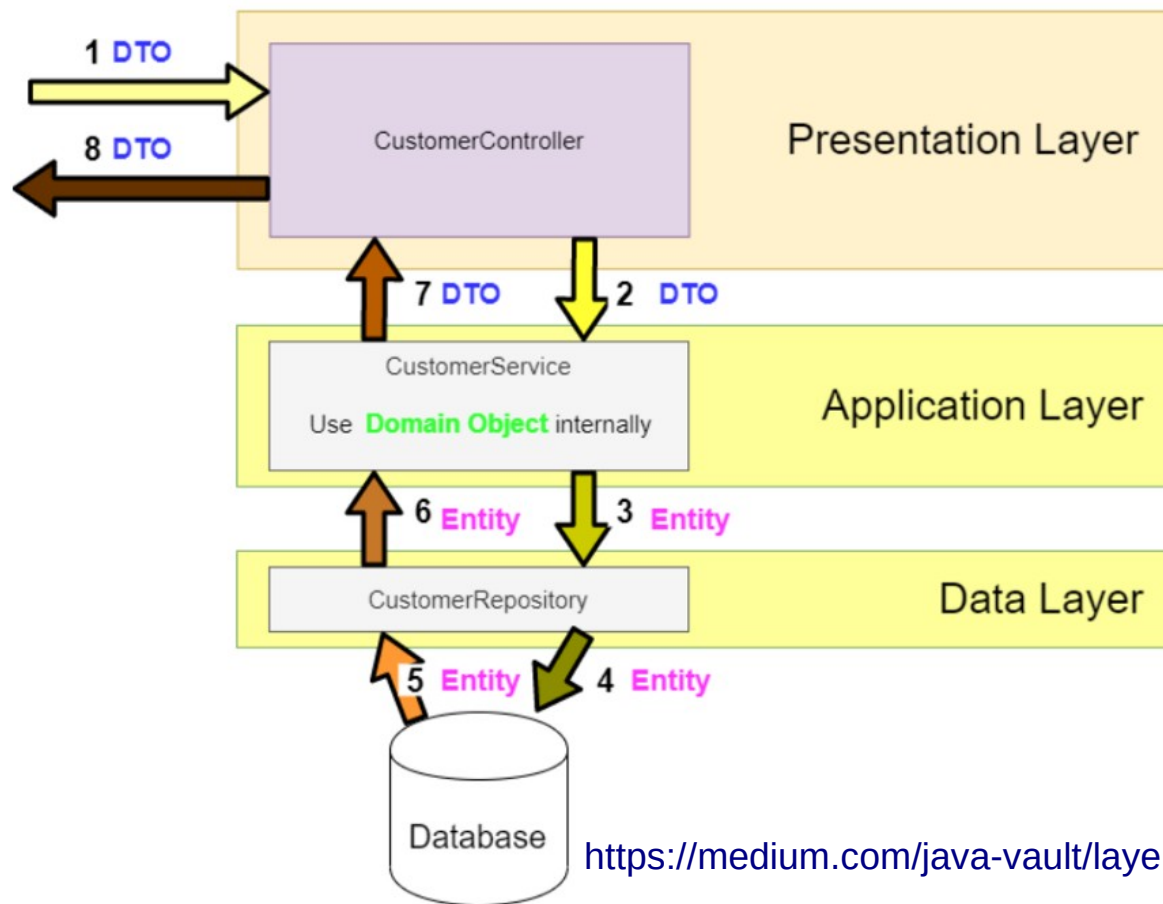
# Spring Cloud Netflix vs. Kubernetes

- Poznámka: Toto není úplně férové srovnání, Kubernetes toho dělá daleko víc, ale toto srovnání má význam pro lidi co znají Spring Cloud Netflix a plánují přecházet na Kubernetes:
  - <https://dzone.com/articles/deploying-microservices-spring-cloud-vs-kubernetes>
  - <https://stackoverflow.com/questions/42435307/spring-cloud-netflix-vs-kubernetes>

# Evoluce web. aplikací v Javě



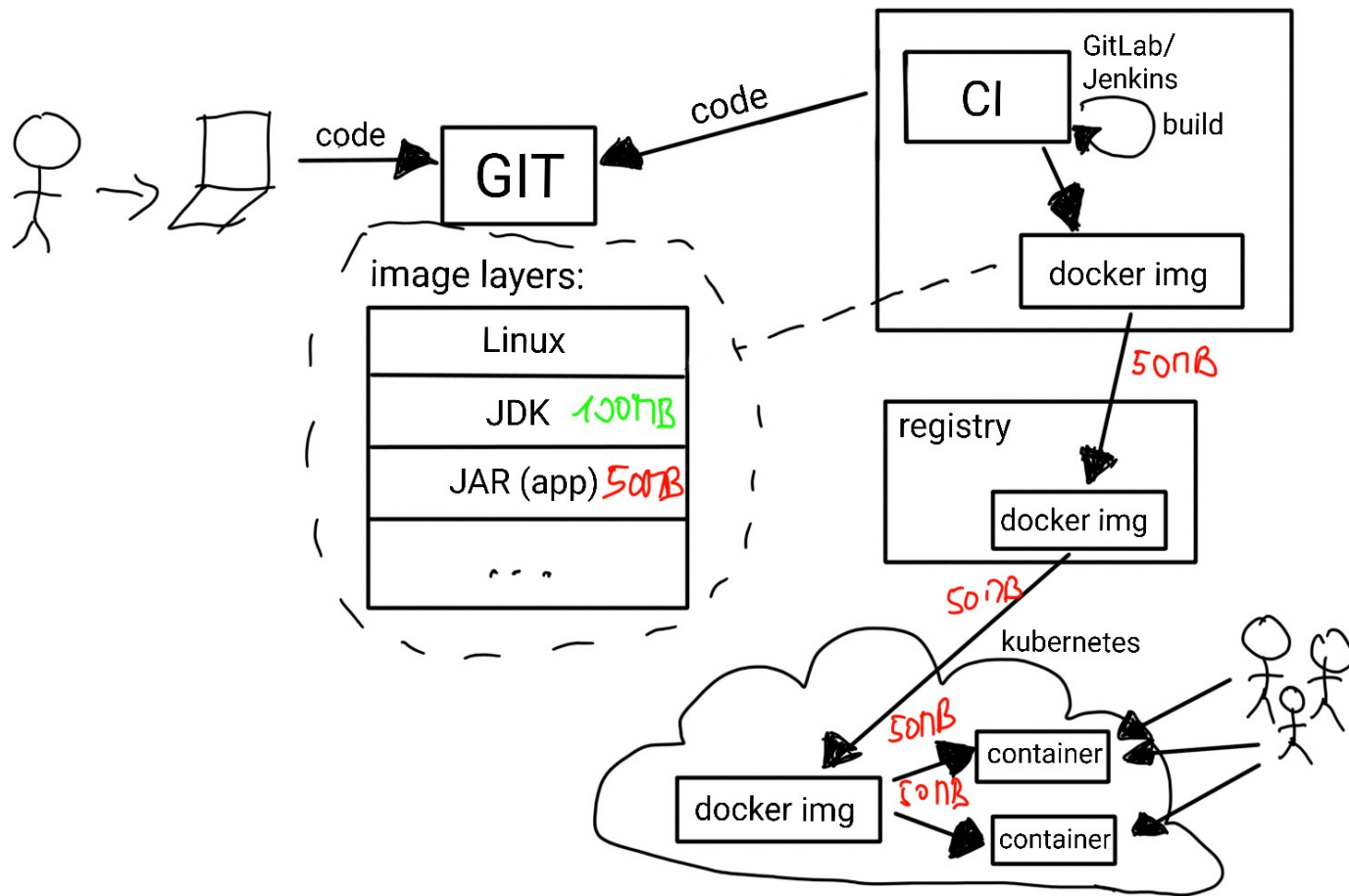
# Třívrstvá architektura



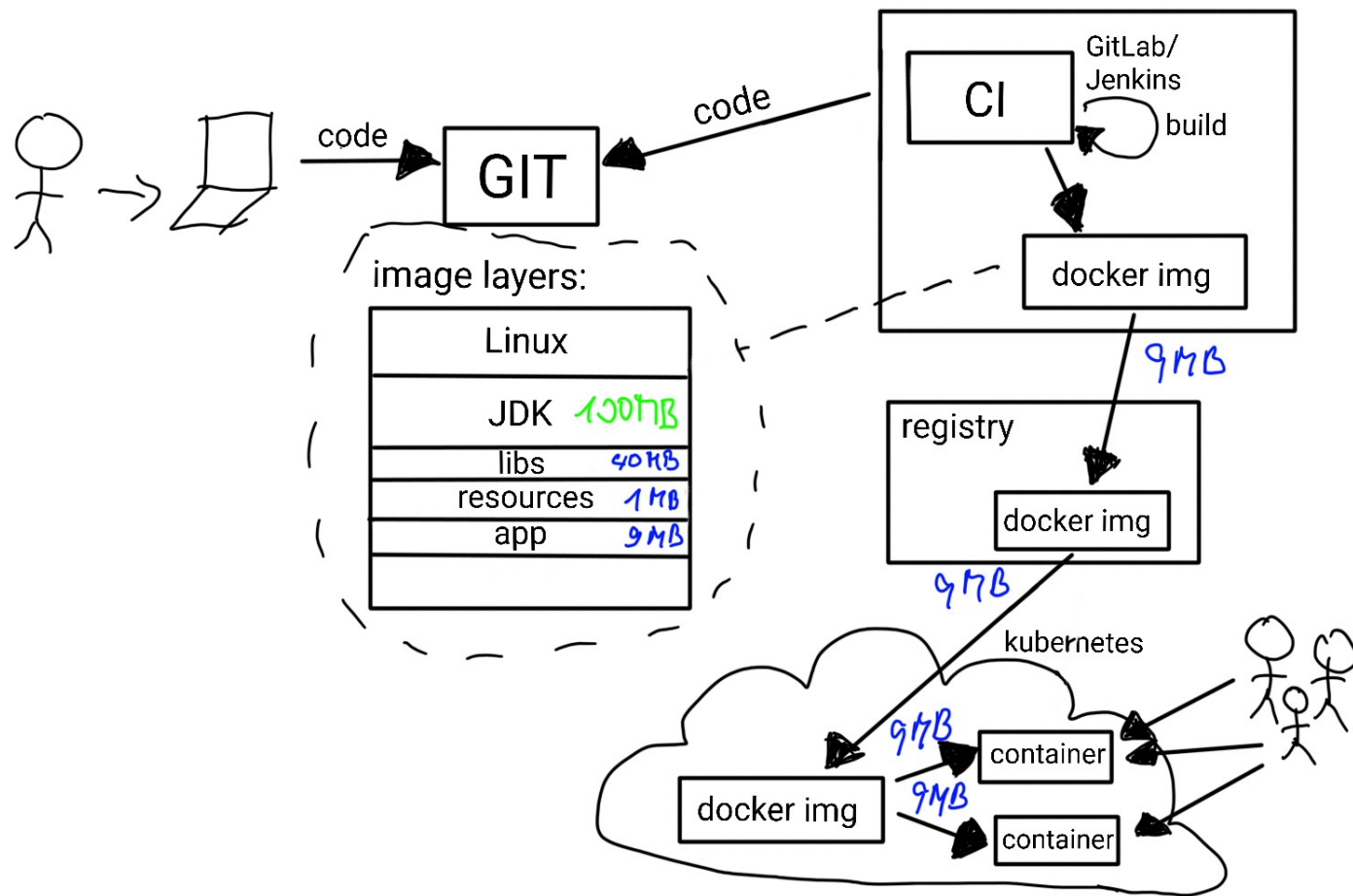
**Poznámka:**  
V Application (Service) Layer se dělá transformace entit na DTO a obráceně pomocí MapStruct (dřív: Dozer, Orika)



# Kód → Kubernetes jeden JAR soubor



# Kód → Kubernetes více-vrstvá image



# Kód → Kubernetes více-vrstvá image

- Jak na to se Spring Boot?
  - Buď pomocí JIB pluginu:
    - <https://github.com/GoogleContainerTools/jib>
  - Nebo od Spring Boot 2.3:
    - <https://spring.io/blog/2020/08/14/creating-efficient-docker-images-with-spring-boot-2-3>

Čtení (asynchronní čtení dat)

# Transakce

- V klasické relační databázi se transakce splňují ACID (Atomicity, Consistency, Isolation, Durability). U microservice se používají pouze lokální transakce, které ACID splňují, ale jenom lokálně.
- U microservices se dále používá princip “Eventual Consistency”
  - Méně striktní než ACID
  - Když se změní data a chvíli se počká, tak všechny read operace vrátí poslední změněnou hodnotu:
    - [https://en.wikipedia.org/wiki/Eventual\\_consistency](https://en.wikipedia.org/wiki/Eventual_consistency)
    - <https://stackoverflow.com/questions/10078540/eventual-consistency-in-plain-english>
  - Proč Eventual Consistency? Daleko lepší škálovatelnost

# Dual Write, Two Phase Commit

- Dual Write je špatný nápad
  - <https://thorben-janssen.com/dual-writes/>
- Two Phase Commit (také známé jako “distribuovaná nebo globální transakce”, používalo se u aplikačních serverů)
  - <https://www.baeldung.com/transactions-across-microservices>
  - V microservisovém světě také špatný nápad
    - <https://thorben-janssen.com/distributed-transactions-microservices/>

# Asynchronous Data Replication I.

- Event driven architecture
  - Microservice (A), které se změnily data, pošle do Kafky (nebo jiné fronty) informaci o změně a změněné hodnotě.
  - Microservice (B), která obsahuje kopii databáze microservic (A) čte data z Kafky a přečte změněnou hodnotu a uloží její hodnotu lokálně.
- Výhody:
  - Microservic mezi sebou nemají žádné závislosti, asynchronní komunikace
  - Jednoduché na změny
- Nevýhody:
  - Hromada schované complexity v Kafce / infrastruktuře, vazby mezi microservicami nejsou na první pohled zřejmé

# Asynchronous Data Replication II.

- Event driven architecture:
  - Dá se použít pro replikaci dat
  - Ale také pro spuštění komplexní business operace
- Domain Event Pattern
  - Jak v microservice (A) uložit data do databáze a publikovat event do Kafky tak, aby bylo zaručeno, že obojí buď projde, nebo neprojde (atomicita celé operace)? Nechceme použít Two Phase Commit. Jak to vyřešit? Pomocí Outbox Pattern a Debezium projektu.
    - <https://microservices.io/patterns/data/transactional-outbox.html>



# Outbox Pattern

- Výborný pro eventually-consistent update
- Jsou možné dvě implementace Message Relay Service:
  1. Provádí polling na outbox tabulku & vytváří a posílá zprávu do Message Broker
    - <https://microservices.io/patterns/data/polling-publisher.html>
  2. Debezium: Monitoruje databázový log & vytváří a posílá zprávu do Message Broker (tzv. Change Data Capture proces – CDC)
    - <https://microservices.io/patterns/data/transaction-log-tailing.html>
- <https://dzone.com/articles/implementing-the-outbox-pattern>

# Outbox Table

- Typický příklad Outbox Table:

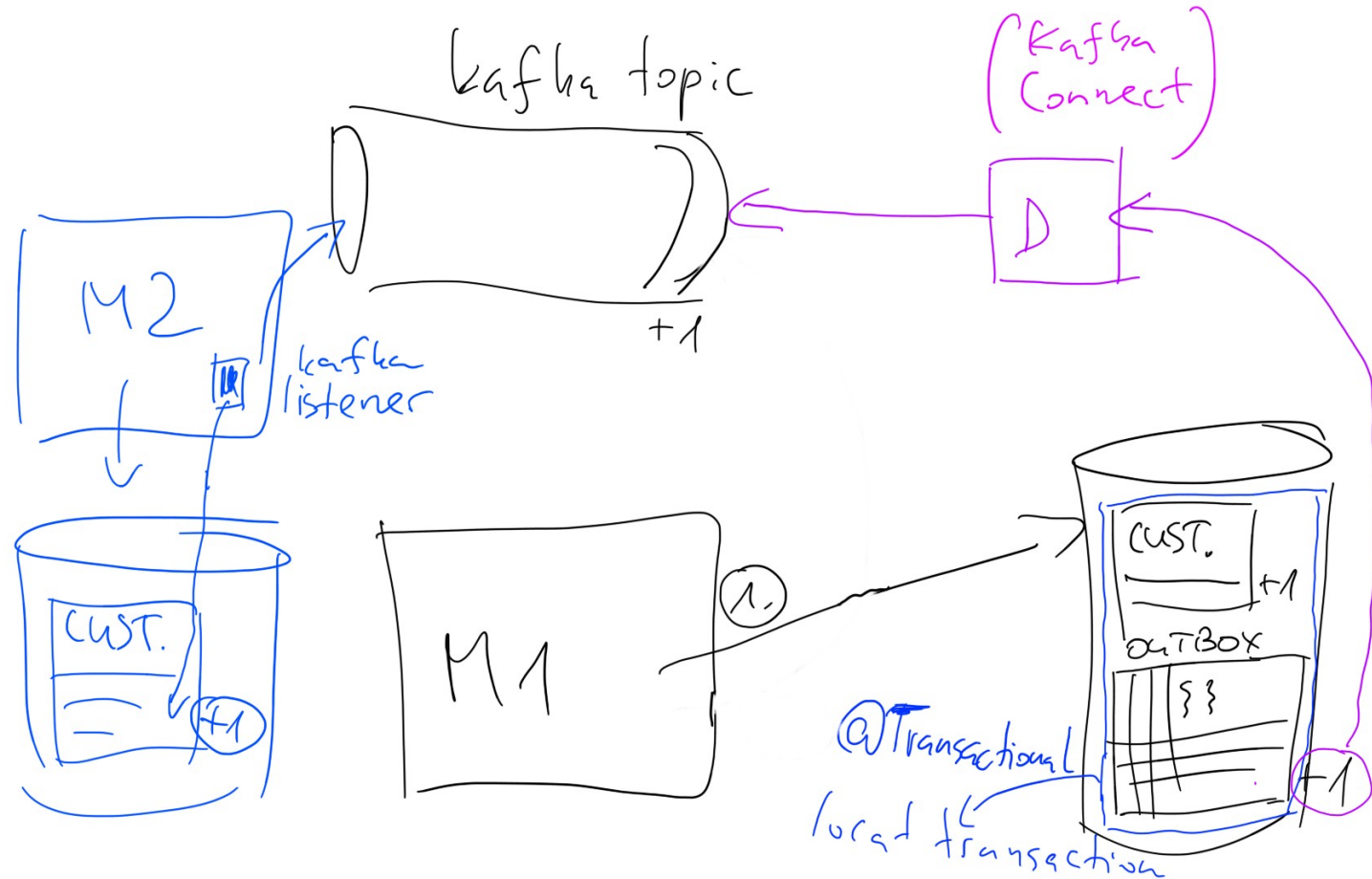
	id [PK] uuid	aggregatetype character varying	aggregateid character varying	type character varying	payload jsonb
1	66c647bb-7aa3...	Book	1	CREATE	{"id": 1, "title": "Hibernate Tips - More than 70 solutions to..."}
2	d4b8d871-94f8...	Book	1	UPDATE	{"id": 4, "title": "Hibernate Tips - More than 70 solutions to..."}

- Zápis do Outbox Table je jednoduchý, může se napsat insert ručně, nebo použít Hibernate / JPA listener / event (nejjednodušší je napsat ručně insert).

# Debezium

- Change Data Capture (CDC):
  - Monitoruje log změn v databázi a publikuje je do Kafky
  - Specifický Kafka Connect connector (od tvůrců Debezium projektu) pro každou databázi. Podporované databáze:
    - MySQL, MongoDB, PostgreSQL, Oracle, SQL Server, Cassandra

# Replikace Dat



Zápis (eventually-consistent transaction)

# SAGA Pattern I.

- Cíl:
  - Zajistit konzistenci dat
  - Vyhnout se použití distribuovaných / globálních transakcí
  - Minimalizovat coupling mezi servisami
- Poznámka:
  - Většinou je lepší spojit microservicery dohromady a vyhnout se použití SAGA patternu.
- <https://microservices.io/patterns/data/saga.html>

# Testování

- Výborný článek ohledně testování Spring Boot aplikací:
  - <https://spring.io/guides/gs/testing-web/>
- Další výborný článek ohledně testování obecně:
  - <https://phauer.com/2019/modern-best-practices-testing-java/>
- Mockování microservis:
  - <http://wiremock.org/>
- Testcontainers pro jednoduché spouštění Docker kontejnerů v testech:
  - <https://www.testcontainers.org/>
- Load testing:
  - Apache JMeter, Gatling, BlazeMeter

# APM / Tracing

- Užitečné když nás zajímá trasování komunikace mezi mikroservisami:
  - Elastic APM
    - Nejjednodušší na nastavení
  - Zipkin / Jaeger
    - Traefik proxy server má integraci s Jaeger:
      - <https://docs.traefik.io/observability/tracing/jaeger/>
    - Spring Boot aplikace se dá také lehce integrovat:
      - <https://medium.com/@klaus.dobbler/introducing-distributed-tracing-to-a-docker-swarm-landscape-f92c033e36d>
    - Jinak se musí programově vytvářet SPANy:
      - <https://www.scalyr.com/blog/jaeger-tracing-tutorial/>



# Jaeger & OkHttp

```
<dependency>
  <groupId>io.opentracing.contrib</groupId>
  <artifactId>opentracing-spring-jaeger-web-starter</artifactId>
  <version>3.1.2</version>
</dependency>
<dependency>
  <groupId>io.opentracing.contrib</groupId>
  <artifactId>opentracing-okhttp3</artifactId>
  <version>3.0.0</version>
</dependency>
```

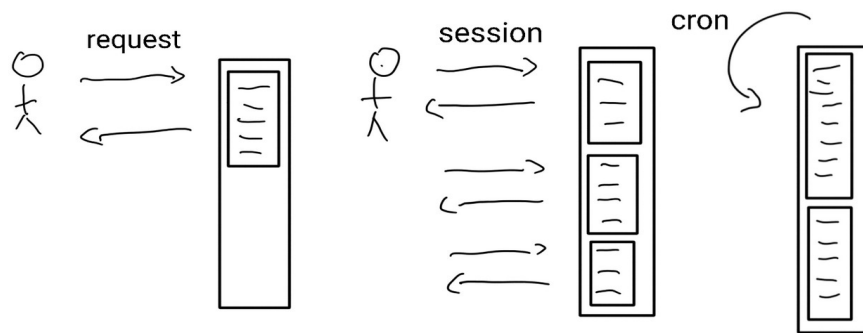
```
opentracing.jaeger.service-name=java-skoleni
opentracing.jaeger.enabled=true
opentracing.jaeger.udp-sender.host=localhost
```

```
OkHttpClient okHttpClient = new OkHttpClient.Builder()
    .connectTimeout(Duration.ofSeconds(5))
    .readTimeout(Duration.ofSeconds(5))
    .build();
Call.Factory client = new TracingCallFactory(okHttpClient, GlobalTracer.get());

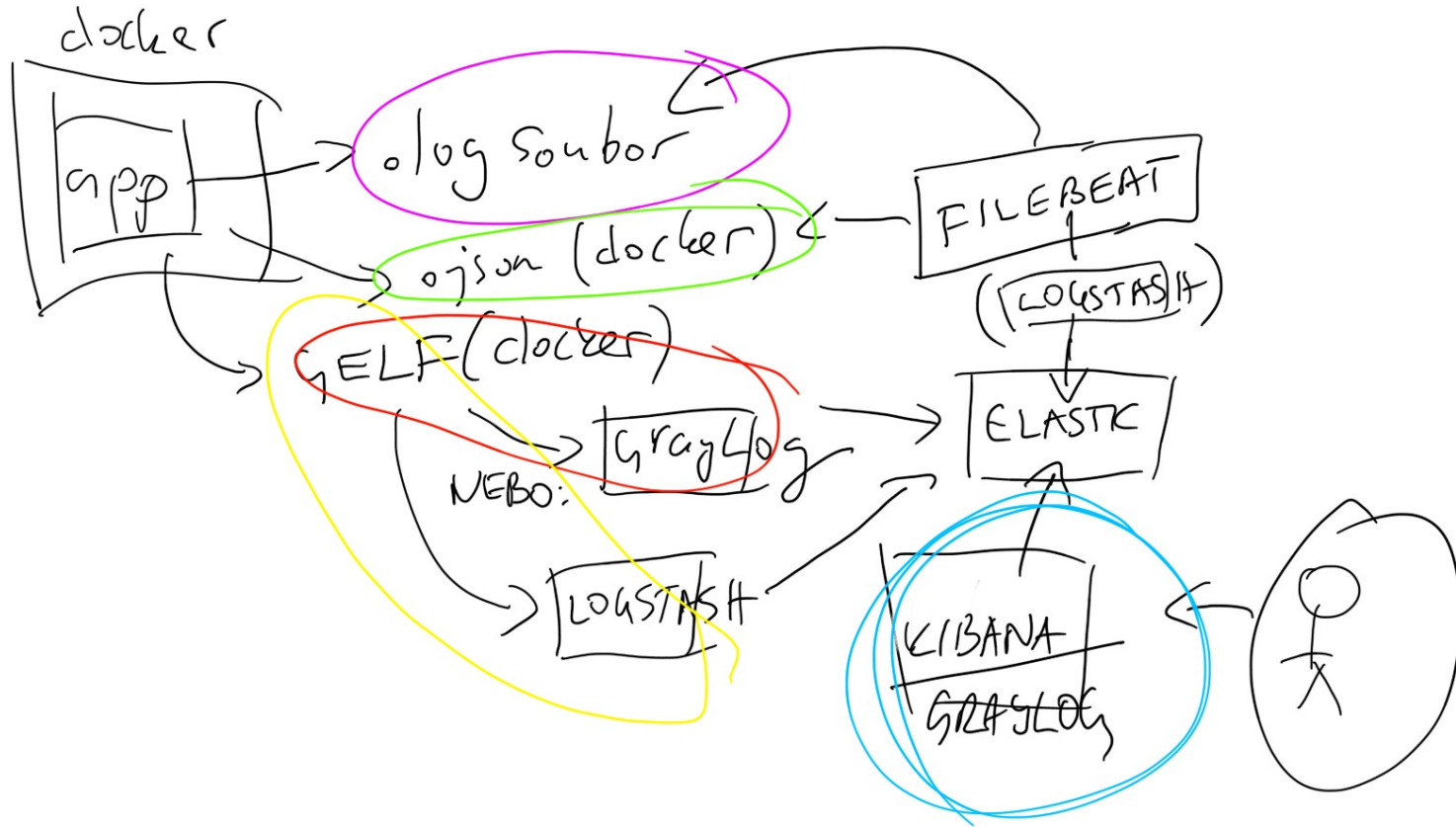
Request = Request.Builder()
    .get()
    .url(url)
    .build();
client.newCall(request).execute();
```

# Microservices: Logování

- Při logování více microservice nebo u reaktivních microservice je užitečné do logu přidat MDC (Mapped Diagnostic Context):
  - <https://dzone.com/articles/mdc-better-way-of-logging-1>
  - <https://spring.io/projects/spring-cloud-sleuth>
- Jinak pro analýzu logů obecně (včetně nebo i bez MDC) je velice užitečný například ELK stack (ElasticSearch & Kibana).
  - Nebo Graylog:
    - <https://www.graylog.org/>
  - Nebo Loki:
    - <https://grafana.com/oss/loki/>



# ELK stack



# Spring Boot Admin I.

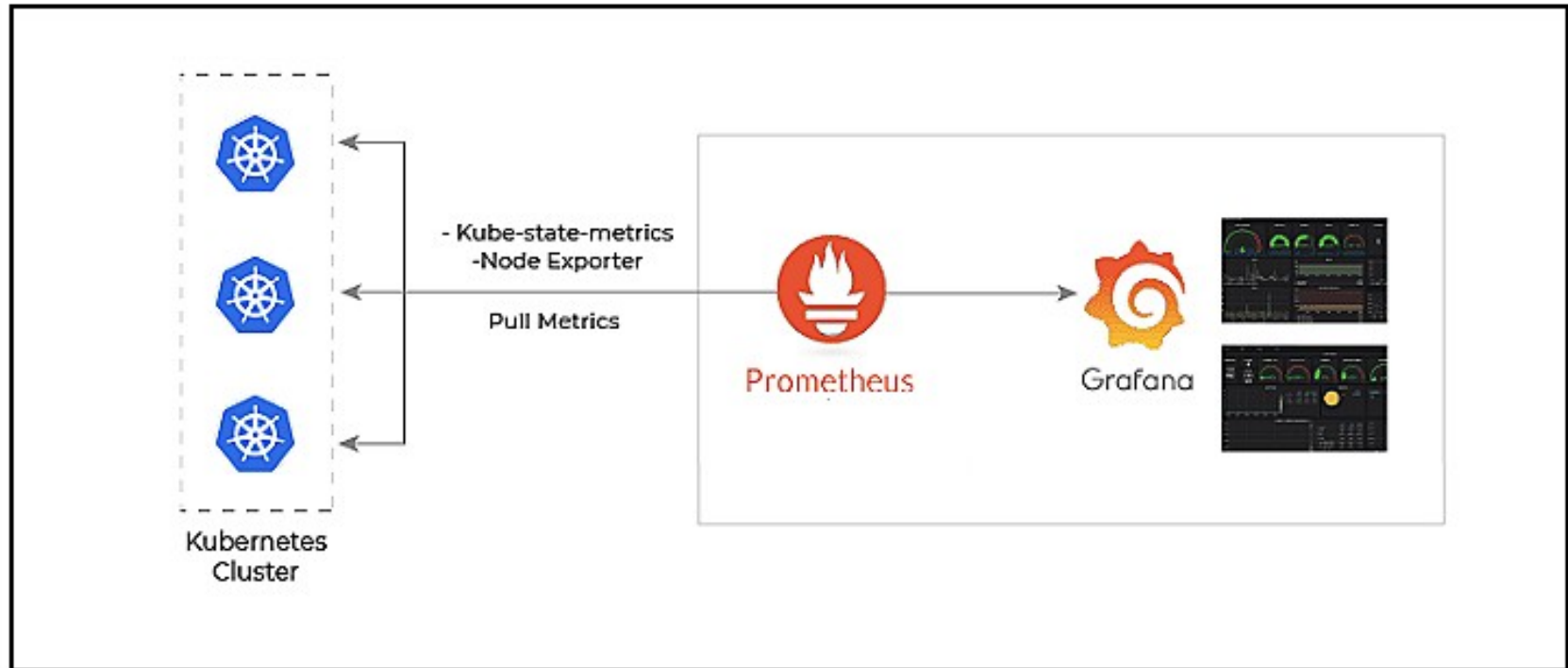
- Pro ovládání Spring Boot aplikací (nastavení logování apod.):
  - <https://github.com/codecentric/spring-boot-admin>
  - Nastavení security:
    - <https://www.vojtechruzicka.com/spring-boot-admin/>
  - Aktivování Log Viewer:
    - <https://chiranjeevigk.wordpress.com/2019/12/11/spring-boot-admin-with-log-viewer/>
    - V případě použití logback.xml je ještě zapotřebí nastavit autoScan:
      - <http://logback.qos.ch/manual/configuration.html#autoScan>
  - Když se u klienta přidá Jolokia a nastaví se `spring.jmx.enabled=true`, tak funguje i export JMX

# Spring Boot Admin II.

- Také je možné aktivovat základní trasování (ale k tomu je lepší Jaeger nebo Zipkin)
  - <https://juplo.de/actuator-httptrace-does-not-work-with-spring-boot-2-2/>
- Jak aktivovat audit events:
  - <https://stackoverflow.com/questions/61298875/spring-actuator-auditevents-endpoint-returns-404>
- Dále je možné aktivovat:
  - Liquibase / Flyway databázové migrace
  - Email notifikace při změně stavu
  - Přehled a mazání aktivních session (se spring session)

# Metrics

- Prometheus + Grafana :-)



# OpenAPI

- Pro dokumentování REST API se používá OpenAPI:
  - <https://springfox.github.io/springfox/>
  - <https://github.com/springdoc/springdoc-openapi>

# Micrometer

- Micrometer: něco jako SLF4J, ale pro metriky (metrics) monitorování aplikace:
  - <https://micrometer.io/>
- Extras – zobrazují jak hodně paměti aplikace používá (funguje jenom na Linuxu a Hotspotu):
  - <https://github.com/mweirauch/micrometer-jvm-extras>
- Custom metrics:
  - [https://www.atlantbh.com/blog/custom\\_metrics\\_micrometer\\_prometheus\\_spring\\_boot\\_actuator/](https://www.atlantbh.com/blog/custom_metrics_micrometer_prometheus_spring_boot_actuator/)



# Micrometer Dashboard

- Velice pěkný Grafana dashboard pro Micrometer (pouze Hotspot):
  - <https://grafana.com/grafana/dashboards/4701>

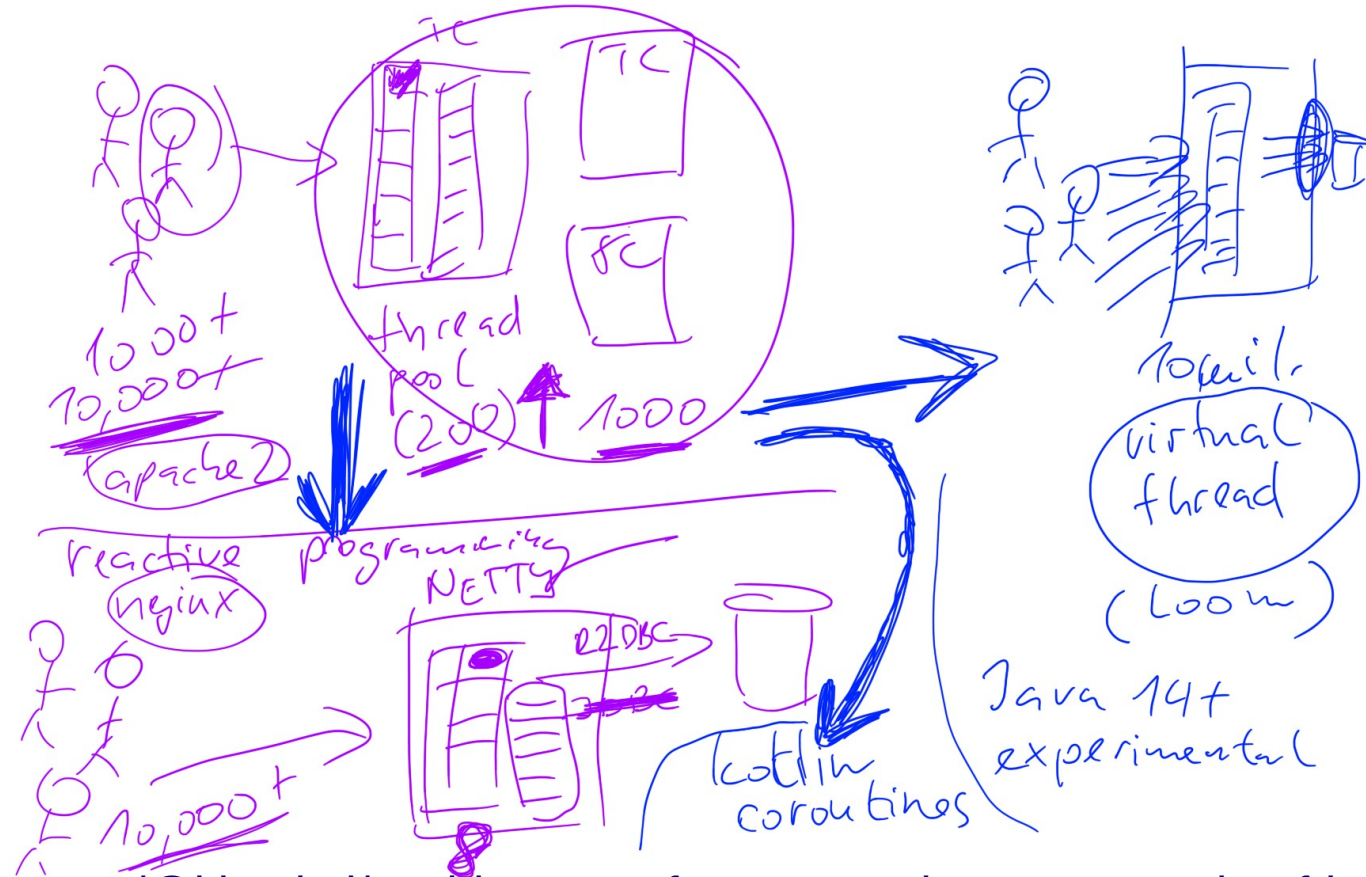
# Alerting + delays

- Pěkný rozbor jak se počítají delays u alertů:
  - <https://pracucci.com/prometheus-understanding-the-delays-on-alerting.html>

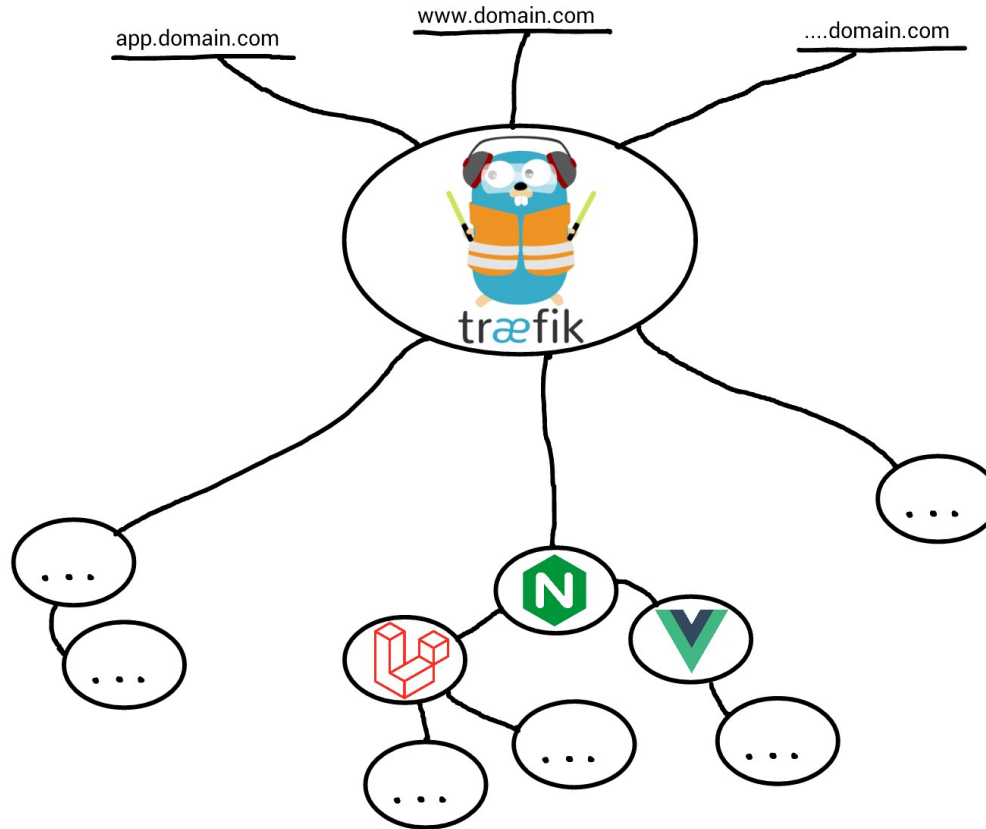
# Prometheus exportery

- Node exporter (je v Ubuntu repozitari, bezi na portu 9100):
  - <https://prometheus.io/docs/guides/node-exporter/>
- Existuji i dalsi jako postgres exporter (bezi na portu 9187):
  - [https://github.com/wrouesnel/postgres\\_exporter](https://github.com/wrouesnel/postgres_exporter)
- Pro Docker & Kubernetes:
  - <https://github.com/google/cadvisor>
- <https://prometheus.io/docs/instrumenting/exporters/>

# Reactive programming / Virtual Threads / ...



# CORS



<https://blog.bitsrc.io/how-and-why-you-should-avoid-cors-in-single-page-apps-db25452ad2f8>

# Service Mesh

- <https://www.tomaskubica.cz/post/2019/kubernetes-prakticky-role-service-mesh/>