JDESIGN

Component-oriented programming

- Component-oriented programming je způsob programování, kdy se nějaký problém rozdělí do sekcí (komponent). V dnešní době se spíš nahrazuje microservices.
- Příklady komponentového programování:
  - JDBC – komponenta, skládající se z řady tříd, která slouží pro práci s databází. Jak to vnitřně dělá nás ale nezajímá.
  - Servlet – komponenta, která je namapovaná na nějaké URL, získá vstup od klienta a pošle mu požadovaný výstup.
  - EJB / Spring bean – opět komponenty.
  - Spring Boot Starter
    - https://www.baeldung.com/spring-boot-custom-auto-configuration
    - https://www.baeldung.com/spring-import-annotation
- https://stackoverflow.com/questions/4947859/what-is-component-oriented-programming-in-java

Microservices, Modular Monolith

- https://www.marcobehler.com/guides/java-microservices-a-practical-guide
- http://www.kamilgrzybek.com/design/modular-monolith-primer/

Balíčková / modulární struktura projektu

- Velká otázka při tvorbě projektu je balíčková struktura. Lze se setkat se dvěma patterny (to také ovlivňuje modulární strukturu projektu):

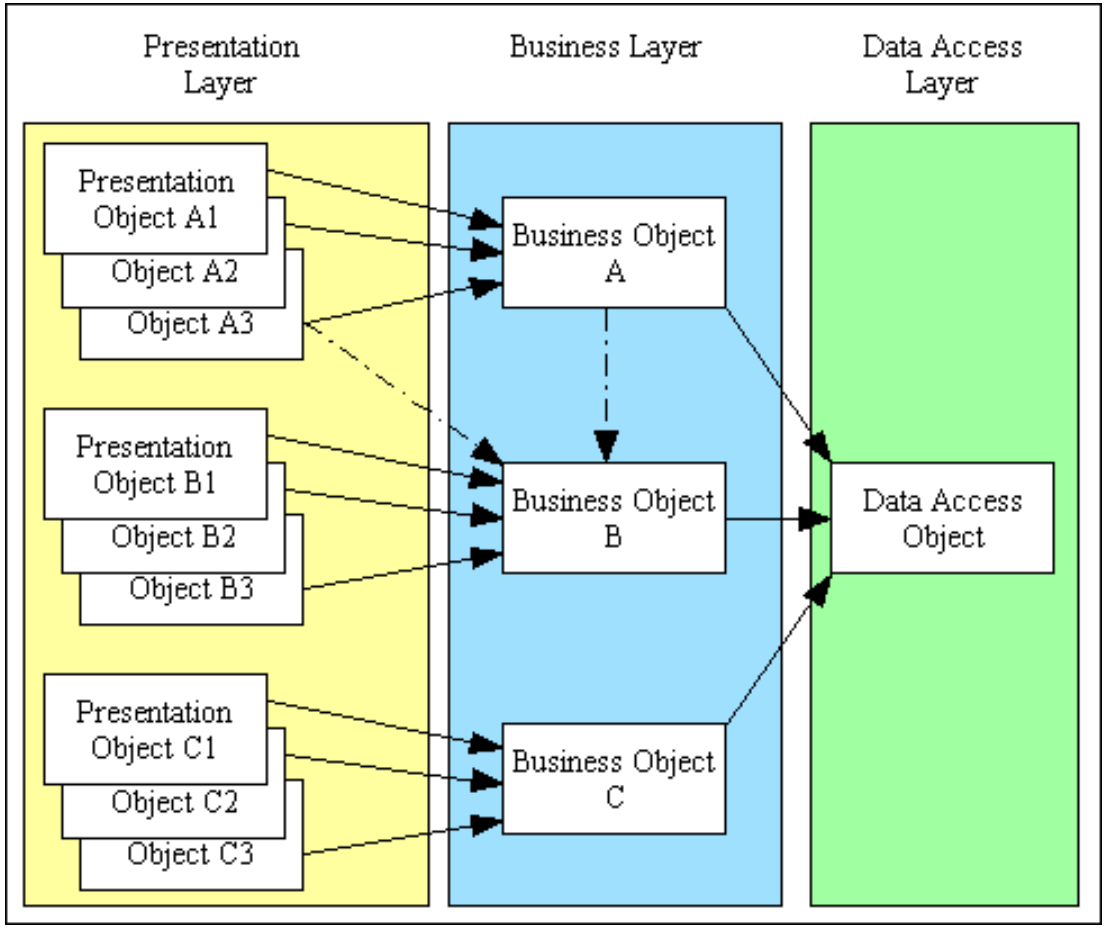
1. Rozdělení podle typu třídy:

```
cz.firma.projekt.entity.Item
cz.firma.projekt.entity.Customer
cz.firma.projekt.repository.ItemRepository
cz.firma.projekt.repository.CustomerRepository
cz.firma.projekt.service.ItemService
cz.firma.projekt.service.CustomerService
```

2. Rozdělení podle use-case / entity:

```
cz.firma.projekt.item.Item
cz.firma.projekt.item.ItemRepository
cz.firma.projekt.item.ItemService
cz.firma.projekt.customer.Customer
cz.firma.projekt.customer.CustomerRepository
cz.firma.projekt.customer.CustomerService
```

Třívrstvá architektura



Poznámka: Z datové vstvy vycházejí entity, v servisní (business) vrstvě je mapper transformuje na DTOčka a ty vrací prezentační vrstva.

Java Bean, POJO, VO, DTO

- Java Bean:
  - Třída, která má konstruktor bez argumentů a k atributům se přistupuje přes gettery / settery
- POJO (Plain Old Java Object):
  - Jednoduchý Java objekt
- Value Object:
  - Objekt, který obsahuje nějakou „jednoduchou“ hodnotu (například Integer nebo Color.RED), VO je immutable.
- DTO (Data Transfer Object):
  - Objekt, který slouží k uchování dat. Například když se načtou data prostřednictvím MyBatis, pak se ukládají do DTO objektů. Nebo se DTO objekty obvykle používají při použití Hibernate a Jackson / JAXB
- https://stackoverflow.com/questions/1612334/difference-between-dto-vo-pojo-javabeans

Singleton

- Pattern, pomocí kterého lze dosáhnout situace, kdy máme v operační paměti pouze jednu instanci (jeden objekt) nějakého typu.
  - https://en.wikipedia.org/wiki/Singleton_pattern
- Je to pattern nebo anti-pattern?
  - IMHO záleží na způsobu implementace:
    - Pokud by se měl singleton implementovat způsobem jako je popsané na wikipedii, pak se jedná o anti-pattern.
    - Pokud se k dosažení cíle mít jeden objekt nějakého typu v operační paměti použije IoC container, pak je „singleton“ pattern.
  - Kdy dělá Singleton problémy? Zejména u testování.

Lazy initialization

- Lazy inicializace není návrhový vzor, ale z pohledu designu se jedná o velice důležitý koncept, kdy se oddaluje vytvoření objektu nebo provedení nějakého náročného procesu do prvního použití.
  - https://en.wikipedia.org/wiki/Lazy_initialization
- Někdy je použití lazy inicializace best practice, jindy přesně naopak.
- Příklady best practices:
  - Použití lazy vazeb u Hibernate

Immutable

- Immutable objekt (objekt, jehož obsah je po jeho konstrukci neměnný – například String) je thread-safe. Pozor ale na to, aby byl objekt skutečně immutable!!!
  - https://dzone.com/articles/do-immutability-really-means
- Vytvořit immutable objekt nemusí být úplně jednoduché. Například první příklad není immutable, protože je možné zvnějšku změnit jeho vnitřní stav:

```
class UnsafeStates {
    private String[] states = new String[] {
        "AK", "AL" ...
    };
    public String[] getStates() { return states; }
}
```

```
@Immutable
public final class ThreeStooges {
    private final Set<String> stooges = new HashSet<String>();

    public ThreeStooges() {
        stooges.add("Moe");
        stooges.add("Larry");
        stooges.add("Curly");
    }

    public boolean isStooge(String name) {
        return stooges.contains(name);
    }
}
```

http://jcip.net/listings/UnsafeStates.java
http://jcip.net/listings/ThreeStooges.java

Thread safety

- Je celá řada objektů, jejichž operace jsou thread-safe a pak je možné používat jednu instanci takového objektu v celé aplikaci.
- Příklady thread-safe objektů:
  - DataSource
  - RestTemplate, JdbcTemplate
  - JAXBContext
  - PrettyTime
    - https://www.ocpsoft.org/prettytime/
    - Pozor! V konstruktoru PrettyTime je logika načítání i18n překladů do paměti, což chvíli trvá (to je obecně špatný návrh). Nicméně vzhledem k tomu, že PrettyTime je thread-safe, tak je možné mít jenom jednu instanci tohoto objektu.
- https://www.baeldung.com/java-thread-safety

Callback, Future, Promise, Reactive programming

- V celé řadě situací je výhodné po skončení metody zavolat callback:
  - https://en.wikipedia.org/wiki/Callback_%28computer_programming%29
- Nebo z metody vrátit Future:
  - https://geekyrui.blogspot.com/2012/06/future-vs-callback.html
- Nebo Promise (v Javě se jmenuje CompletableFuture):
  - https://stackoverflow.com/questions/14541975/whats-the-difference-between-a-future-and-a-promise
- https://medium.com/javascript-in-plain-english/promise-vs-observable-vs-stream-165a310e886f

Observer, EventBus

- Observer je návrhový vzor, ve kterém jeden objekt (subject) obsahuje list na něm závisejících objektů (observers) a při změně stavu subjectu notifikuje observery.
  - https://en.wikipedia.org/wiki/Observer_pattern
- EventBus je prakticky implementace Observer patternu. Existuje několik prakticky používaných implementací v Javě:
  - Guava EventBus
  - greenrobot:eventbus (používané zejména u Android aplikací)

Messaging systems

- V případě, že byste potřebovali publish-subscribe komunikaci mezi servery, pak se podívejte na:
  - JMS
  - RabbitMQ, ActiveMQ apod.
  - Kafka

Observer, RxJava, Project Reactor

- Obdobně je implementací Observer patternu RxJava:
  - https://github.com/ReactiveX/RxJava

Builder

- Návrhový vzor, který slouží pro abstrakci tvorby složitých objektů. Obyčejně bychom mohli použít pro zjednodušení tvorby objektu konstruktor. Ale co když nepotřebujeme vždy nastavovat všechny argumenty? Pak můžeme použít více konstuktorů, ale počet jejich kombinací by byl dřív nebo později neúnosný. A v takové situaci se využije návrhový vzor Builder:
  - https://en.wikipedia.org/wiki/Builder_pattern
  - https://projectlombok.org/features/Builder

Factory method

- Factory method je návrhový vzor, ve kterém metoda vrací nějaký objekt, aniž by se konkrétně přesně specifikovalo jaký typ objektu se má vrátit:
  - https://en.wikipedia.org/wiki/Factory_method_pattern
  - V celé řadě situací je best practice používání těchto metod. Příklad: Integer.valueOf(), List.of(), Path.of()
  - Někdy factory metody navíc z důvodu optimalizace používají pool objektů (když vrací immutable objekty) … příklad: Integer.valueOf()

Strategy

- https://en.wikipedia.org/wiki/Strategy_pattern
- https://refactoring.guru/design-patterns/strategy

Iterator

- Iterator je návrhový vzor, který abstrahuje procházení kolekce záznamů tak, aby nebylo implementačně závislé na použitém druhu kolekce.
  - https://en.wikipedia.org/wiki/Iterator_pattern

Service Locator

- Service Locator je návrhový vzor, který obsahuje centrální registr objektů, ze kterého vrací jednotlivé objekty. Slouží tedy k zapouzdření získání objektů:
  - https://en.wikipedia.org/wiki/Service_locator_pattern
  - Toto není něco, co byste obyčejně chtěli implementovat, ale nachází se v containerech jako je Spring pro získávání referencí na beany (ve Springu to je ApplicationContext):
    - https://stackoverflow.com/questions/22795459/is-servicelocator-an-anti-pattern

Dependency Injection & IoC

- Návrhový vzor Dependency Injection (DI) je základem Springu. V tomto návrhovém vzoru Injector vytváří objekty a následně jejich reference vkládá do atributů jiných objektů.
  - https://en.wikipedia.org/wiki/Dependency_injection
- Inversion of Control (IoC) je architektura, ve které nějaký framework vytváří objekty a nastavuje jejich vazby. Jednou z implementačních technik IoC je DI.
  - https://en.wikipedia.org/wiki/Inversion_of_control#Implementation_techniques

Spring

- Pokud začínáte se Springem v době Spring Bootu, pak bych Vám doporučil tento dokument k pochopení základů Spring Frameworku:
  - https://www.marcobehler.com/guides/spring-framework

Request / Response mapper

- Zejména pokud používáte Hibernate a posíláte klientovi objekty ve formátu JSON (ale i v jiných situacích – například když získáváte data z nějaké webové služby a chcete je poslat klientovi), tak využijete pattern Request / Response mapper.
- Jedná se o třídy, které mají na starosti transformaci POJO na jiná POJO (například entity -> DTO).
- http://www.servicedesignpatterns.com/RequestAndResponseManagement
- Krásnou implementaci tohoto patternu generuje MapStruct:
  - http://mapstruct.org/

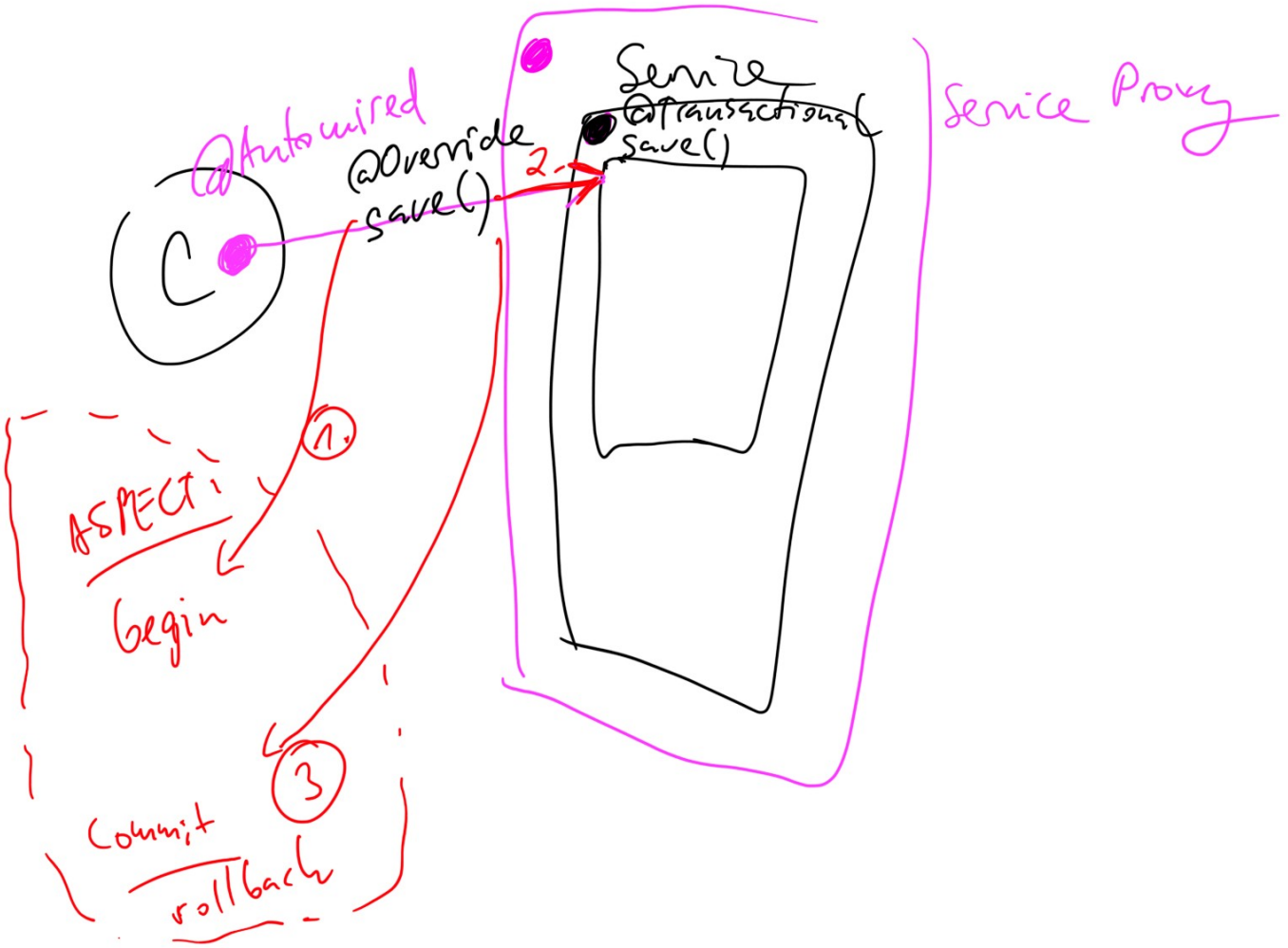
Scope

- Spring implementuje následující scope:
  - Singleton, prototype, request, session (+ je možné vytvořit vlastní scope)
    - http://www.baeldung.com/spring-bean-scopes
- CDI má následující scope:
  - Request, session, application, dependent, conversation
    - http://docs.oracle.com/javaee/6/tutorial/doc/gjbbk.html

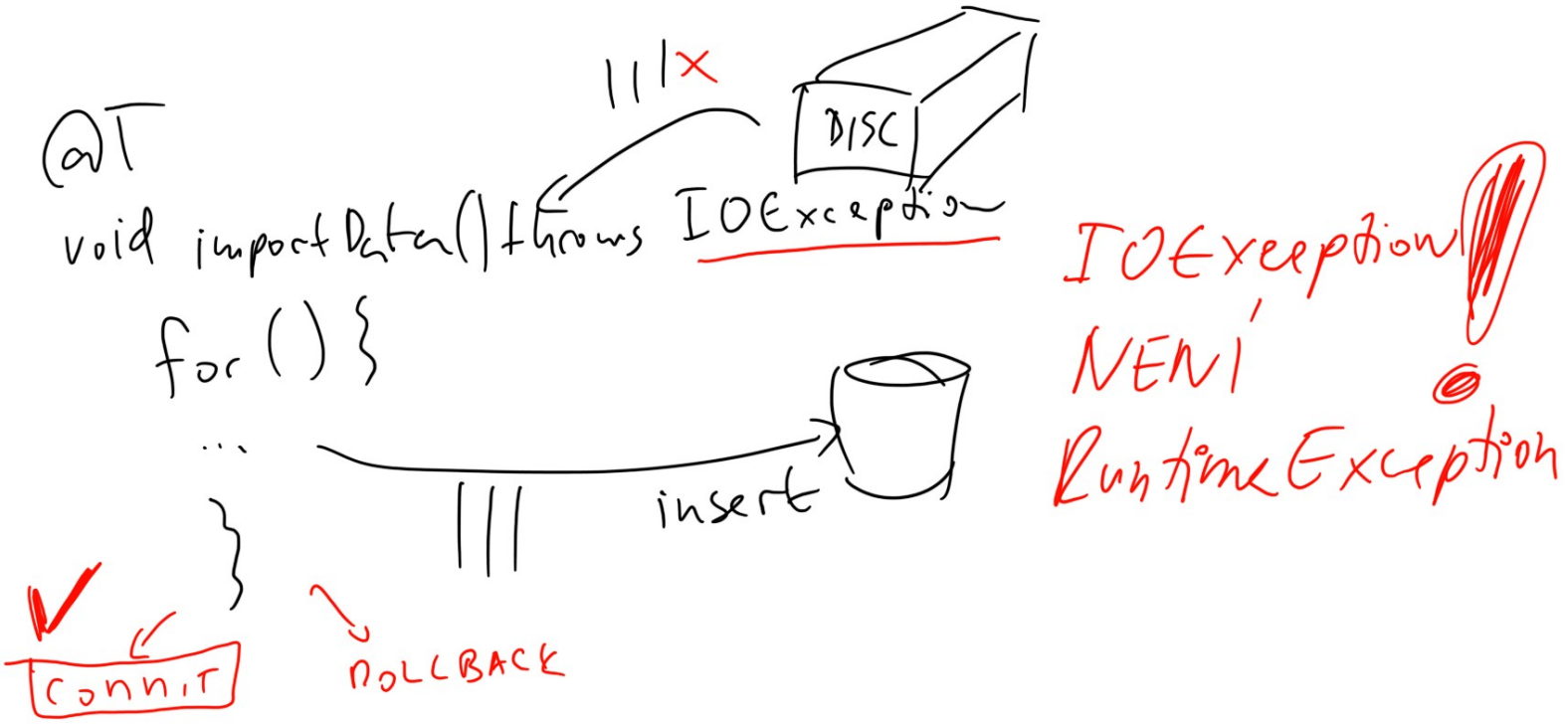
Proxy

- Proxy je třída, která funguje jako interface pro jiný objekt, který „obaluje“ a může změnit funkcionalitu jeho metod.
  - https://en.wikipedia.org/wiki/Proxy_pattern
  - Spring AOP vytváří proxy objekty, které volají logiku, která se zapíná anotacemi @Transactional, @Cacheable, @Async, nebo @PreAuthorize.
  - V Hibernate se proxy používá pro lazy vazby.

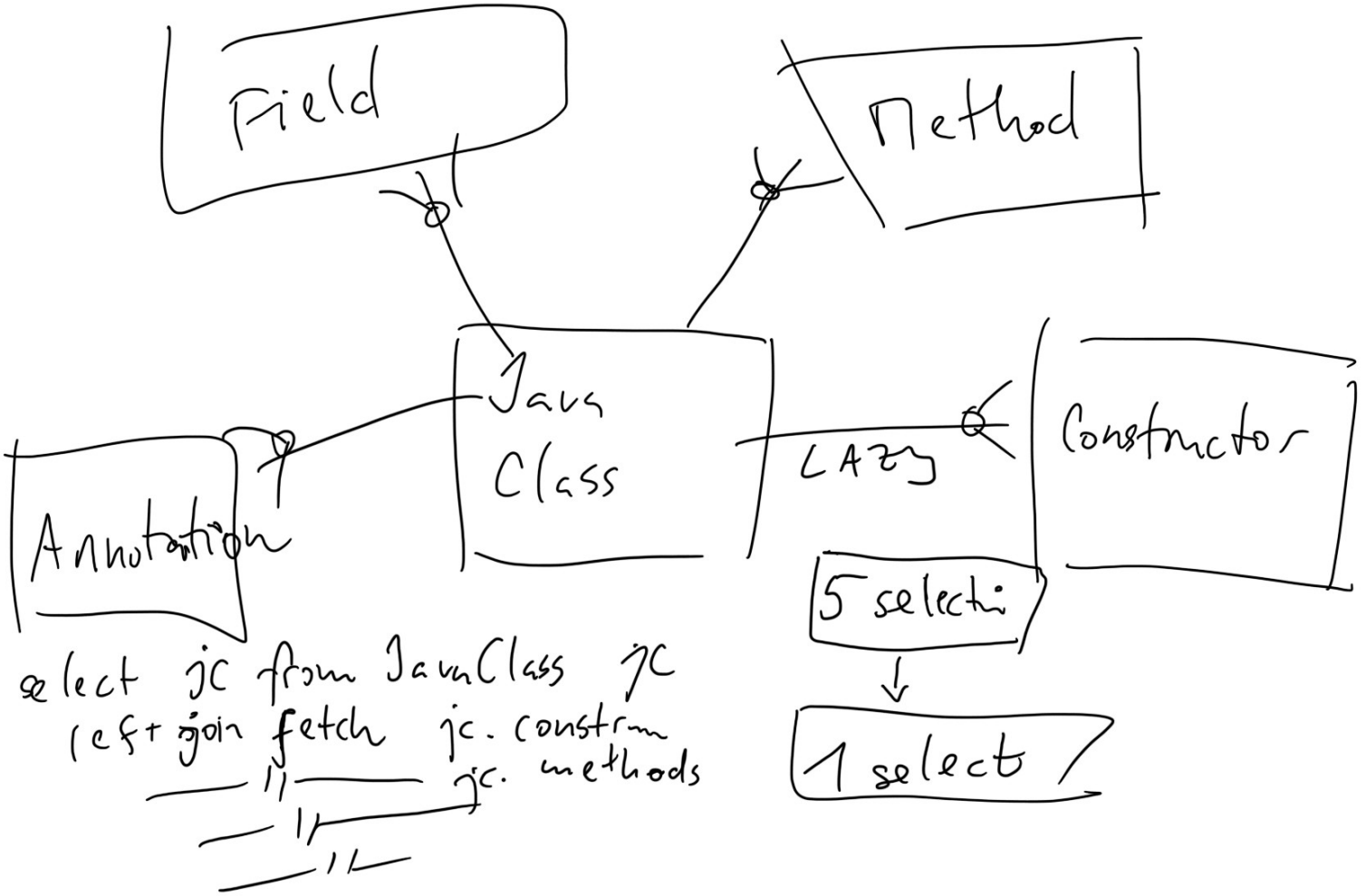
U Springu u AOP pozor na to, abyste přistupovali přes proxy!!!



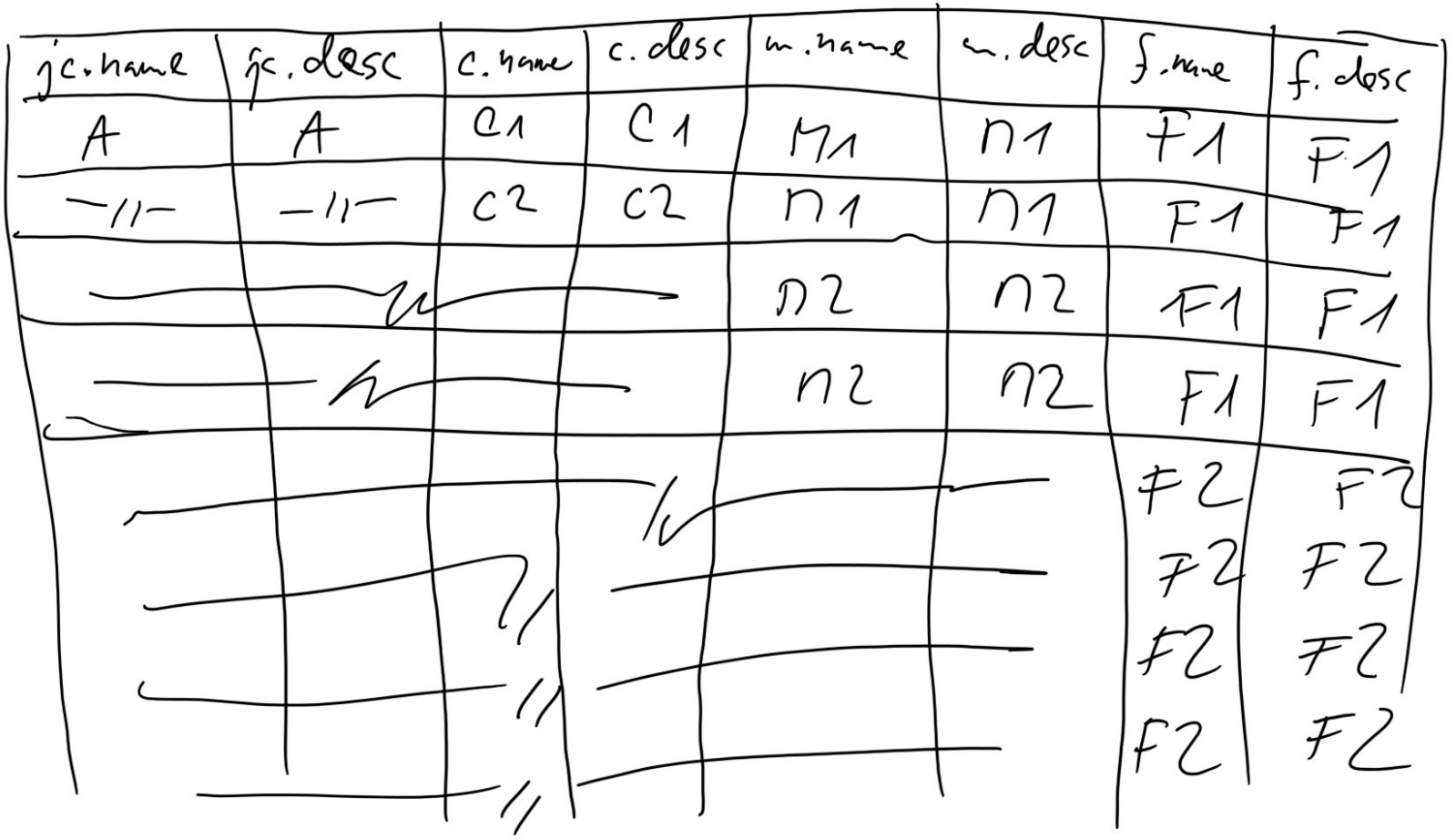
Pozor! Ve výchozím nastavení se transakce rollbackuje pouze při vyhození výjimky typu RuntimeException!



Pozor na left join fetch (nebo entity graph, nebo eager vazbu) u vazby na kolekci entit!!!



Ono to zafunguje, ale … v ResultSetu bude hromada duplicit!



AOP

- Pomocí AOP se vyčlení „cross-cutting concerns“ do aspektů a deklarativně se zapojí na nějaké metody.
  - Typické použití ve Springu: @Transactional, @Cacheable, @Async, @PreAuthorize

Interceptor

- Interceptor automaticky a transparentně zachytává vstupní anebo výstupní požadavky, jedná se tedy o podobný mechanismus jako AOP.
  - https://en.wikipedia.org/wiki/Interceptor_pattern
- Typické použití: javax.servlet.Filter
  - Krásné využití je ve Spring Security

Decorator

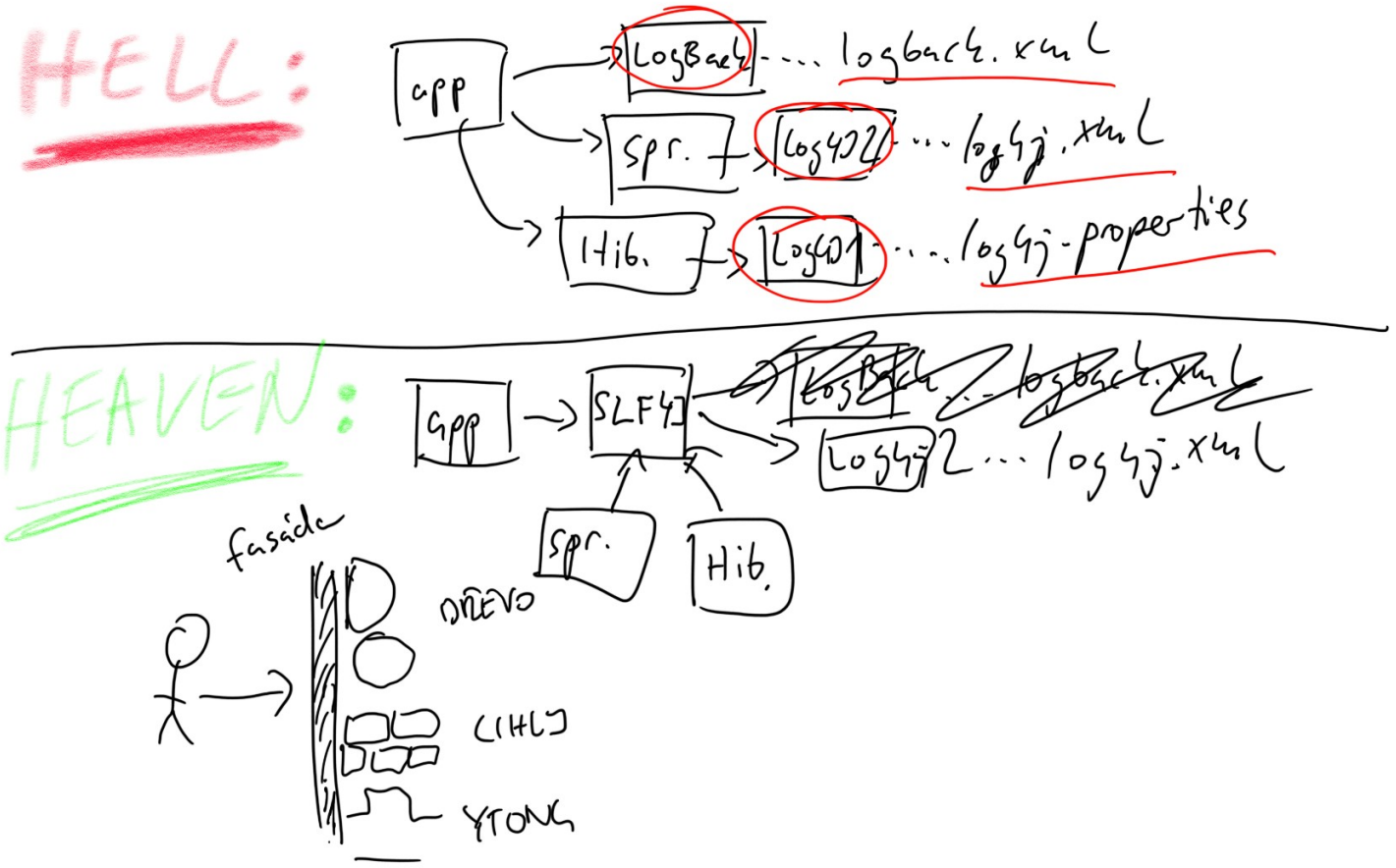
- Decorator slouží k rozšíření funkcionality nějakého objektu:
  - https://en.wikipedia.org/wiki/Decorator_pattern
- Oproti proxy se liší tím, že do decoratoru se vždy vkládá objekt, jehož funkcionalitu bude rozšiřovat, zatímco proxy si ho vytvoří sama.
  - https://stackoverflow.com/a/25195848/894643

Adapter

- Adapter umožňuje, aby bylo možné použít interface stávající třídy jako jiný interface. Prakticky umožňuje, aby dva nekompatibilní interface mohly spolupracovat.
  - https://en.wikipedia.org/wiki/Adapter_pattern
  - https://refactoring.guru/design-patterns/adapter

Facade

- Facade odstiňuje jednu část systému od jiné. Nejčastěji používaná Facade: SLF4J :-)
  - https://en.wikipedia.org/wiki/Facade_pattern



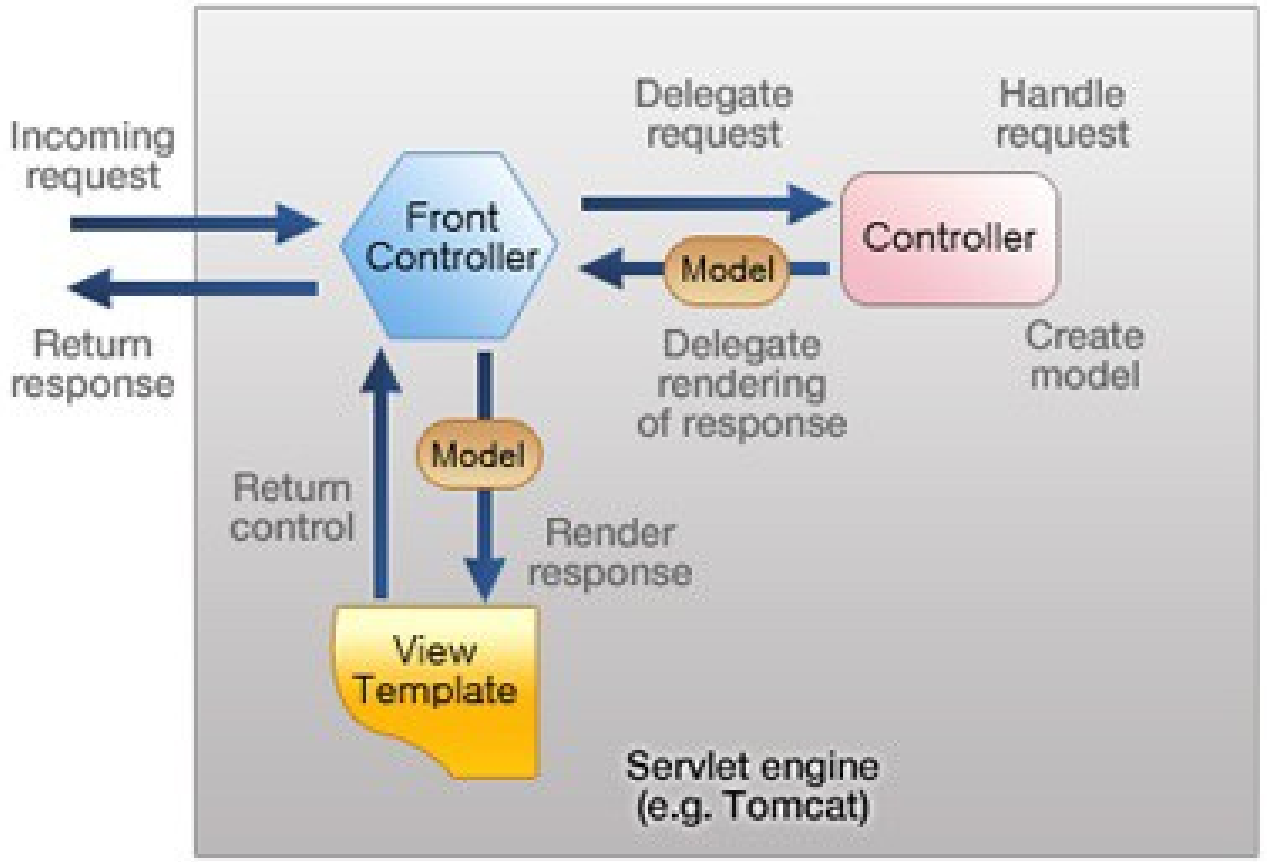
Pool objektů

- V celé řadě situací potřebujeme poolovat nějaké objekty:
  - Vlákna
    - http://tutorials.jenkov.com/java-util-concurrent/executorservice.html
  - Připojení do databáze
    - https://github.com/brettwooldridge/HikariCP
  - Pozor! Vlastní pool objektů byste měli vytvářet pouze, když víte co děláte :-)
    - https://softwareengineering.stackexchange.com/questions/115163/is-object-pooling-a-deprecated-technique

Open In View (Anti)pattern

- Pozor! Tento mechanismus je ve Spring Boot automaticky zapnutý! NEPOUŽÍVAT!!!
  - https://vladmihalcea.com/the-open-session-in-view-anti-pattern/

Front Controller & MVC



Poznámka: MVC (a další návrhové vzory) na straně serveru v současnosti ztrácí smysl, ale na Front Controller je stále velice významný. Ve Spring je Front Controller třída DispatcherServlet.

REST

- REST je architektura, která definuje sadu omezení pro tvorbu webových služeb.
  - https://en.wikipedia.org/wiki/Representational_state_transfer
  - Základ jsou dva typy URL: jedno reprezentuje kolekci záznamů, druhé detail záznamu a GET, POST, PUT, DELETE operace (+ případně PATCH operace).
- PUT vs. PATCH:
  - https://stackoverflow.com/questions/28459418/rest-api-put-vs-patch-with-real-life-examples
- GET operace by měly být idempotent!!!
- Pro dokumentování REST WS slouží OpenAPI:
  - https://springdoc.org/

HATEOAS

- REST je super v tom, že je flexibilní. Ale někdy je zbytečně moc flexibilní … a v takových případech může být vhodné použít HATEOAS. O co se jedná? Je to podmnožina RESTu a základním principem je, že součástí odpovědi serveru jsou další meta-informace (URL), pomocí kterých je možné s dokumentem dělat další operace:
  - https://en.wikipedia.org/wiki/HATEOAS
  - https://spring.io/projects/spring-hateoas

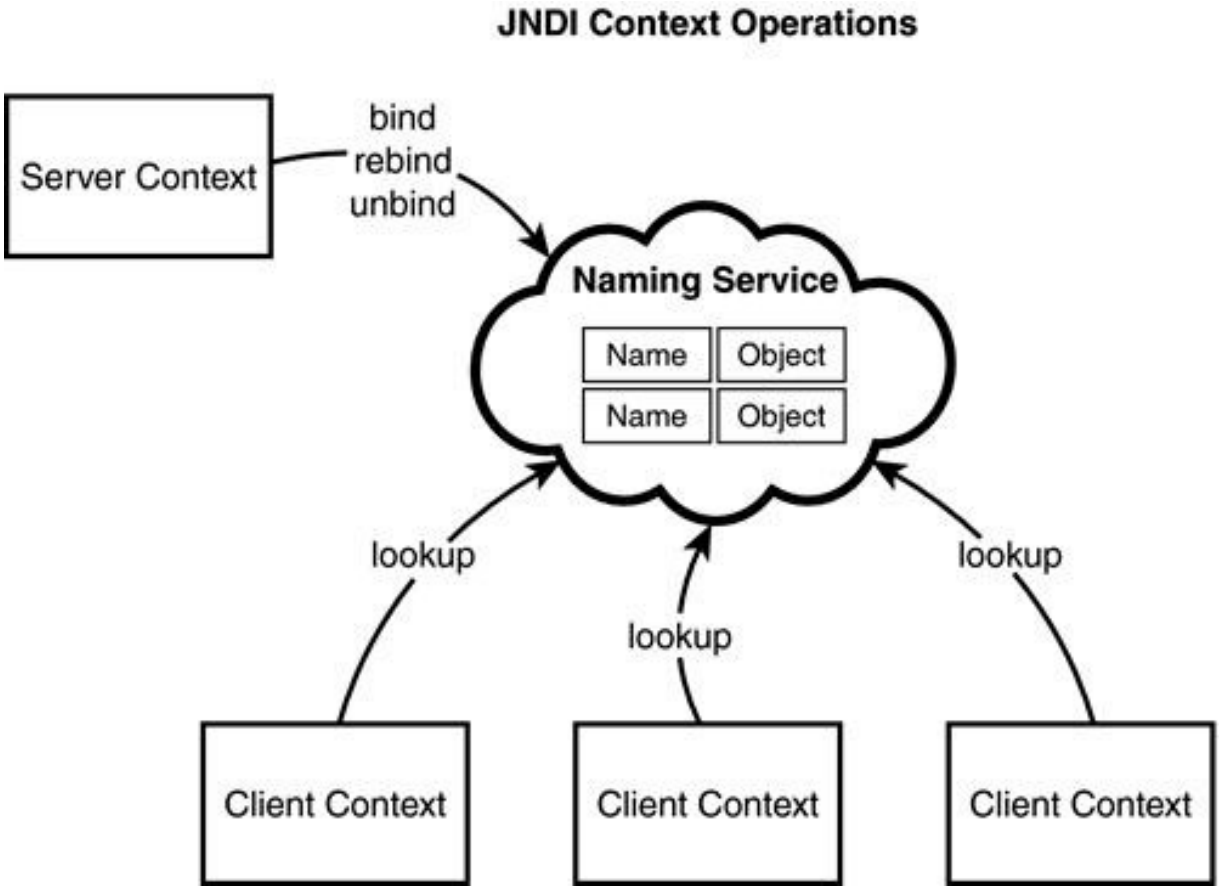
HAL

- HAL (Hypertext Application Language) je implementací HATEOAS a také slouží k tomu, aby v odpovědi serveru byly nejenom samotná data, ale také metainformace, pomocí kterých je možné navigovat na další zajímavé dokumenty:
  - https://en.wikipedia.org/wiki/Hypertext_Application_Language
  - https://stackoverflow.com/questions/25819477/relationship-and-difference-between-hal-and-hateoas
- Projekt postavený na HAL je Spring Data REST:
  - https://spring.io/projects/spring-data-rest
  - Jedná se o výborný nástroj pro generování REST API zejména pro čítelníky.

GraphQL

- REST má tu nevýhodu, že je nutné mít pro každý use-case samostatný endpoit. Čím bude větší množství endpointů a jakmile budeme chtít z důvodu optimalizace posílat přes síť různé podmnožiny entit, tak se vyplatí spíš použít GraphQL:
  - https://graphql.org/
  - GraphQL je prakticky něco jako SQL pro Váš endpoint.

JNDI


JNDI Context Operations
Server Context
bind
rebind
unbind
Naming Service
Name
Object
Name
Object
lookup
lookup
lookup
Client Context
Client Context
Client Context

try-with-resources

- Celá řada tříd implementuje AutoCloseable interface. Pro volání metody close() je best practice používat try-with-resources:
  - https://www.baeldung.com/java-try-with-resources
- Poznámka:
  - V Java 9 try-with-resources doznalo vylepšení:
    - https://dzone.com/articles/try-with-resources-enhancement-in-java-9

Důležité nástroje

- SonarQube (server) / SonarLint (v IntelliJ Idea)
- Gitlab CI / Jenkins / TeamCity

Functional Programming

- http://tutorials.jenkov.com/java-functional-programming/index.html

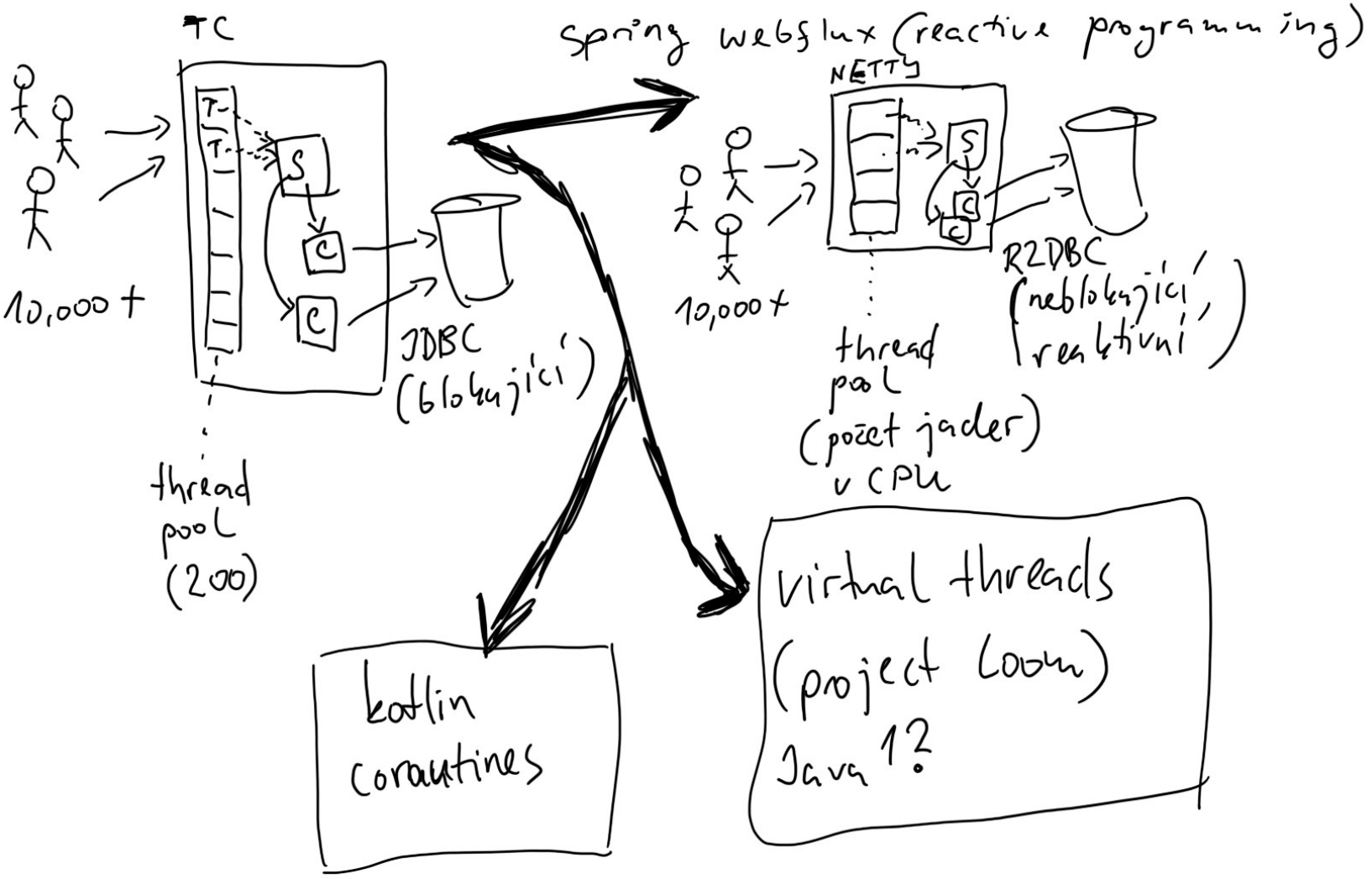
Best Practices I.

- Vlákna:
  - Nepoužívat new Thread(), ale používat Executor (pool vláken)
- V konstruktoru (a getterech / setterech) nesmí být logika, ta patří jinam!!!
- Pro další best practices mohu s klidným srdcem doporučit tyto knihy:
  - Joshua Bloch: Effective Java
  - Vlad Mihalcea: High Performace Java Persistence
  - Chris Richardson: Microservices Patterns
- Jak správně psát JUnit testy:
  - https://phauer.com/2019/modern-best-practices-testing-java/

Lombok atd.

- Pro generování tzv. boilerplate kódu se používá například Lombok, ale měli bychom být s ním opatrní a generovat pouze to, co chceme!!!
  - https://deinum.biz/2019-02-13-Lombok-Data-Ojects-Arent-Entities/
- Další zajímavou knihovnou pro generování boilerplate kódu je:
  - https://github.com/immutables/immutables
    - Slouží pro tvorbu Value Objectů pomocí builder patternu (super pro DTOčka)
  - Nebo AutoValue:
    - https://github.com/google/auto/tree/master/value
- Od Java 14 Records:
  - https://openjdk.java.net/jeps/359
- https://orestkyrylchuk.com/lombok-alternatives

Reactive programming / Virtual Threads / ...



https://medium.com/@kkgulati/avoid-reactor-freeze-reactive-programming-fdc0b4b5991

Event Loop

- U reaktivního programování se používá Event loop princip:

Výhody reaktivního programování (zvýšení throughput a konstantní využití RAM):
https://www.quora.com/What-is-the-better-web-server-stack-Nginx-or-Apache

Různé knihovny pro práci s databází