JDESIGN

Refactoring Guru :-)

- Pro GoF (Gang of Four) design patterns:
  - https://refactoring.guru/design-patterns/

Component-oriented programming

- Component-oriented programming is a programming technique, where some problem is split to sections (components). Nowadays it's often being replaced by microservices.
- Examples:
  - JDBC – component, consisting of many classes, which are used for database connectivity. How it's done internally is not relevant for us, just the interface.
  - Servlet – component, mapped to some URL, which receives client's request and sends back appropriate response.
  - EJB / Spring bean – again, components
  - 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/

Package / modular structure of project

- Really big question when creating any project is package structure. There are two basic patterns (and many variations in between):

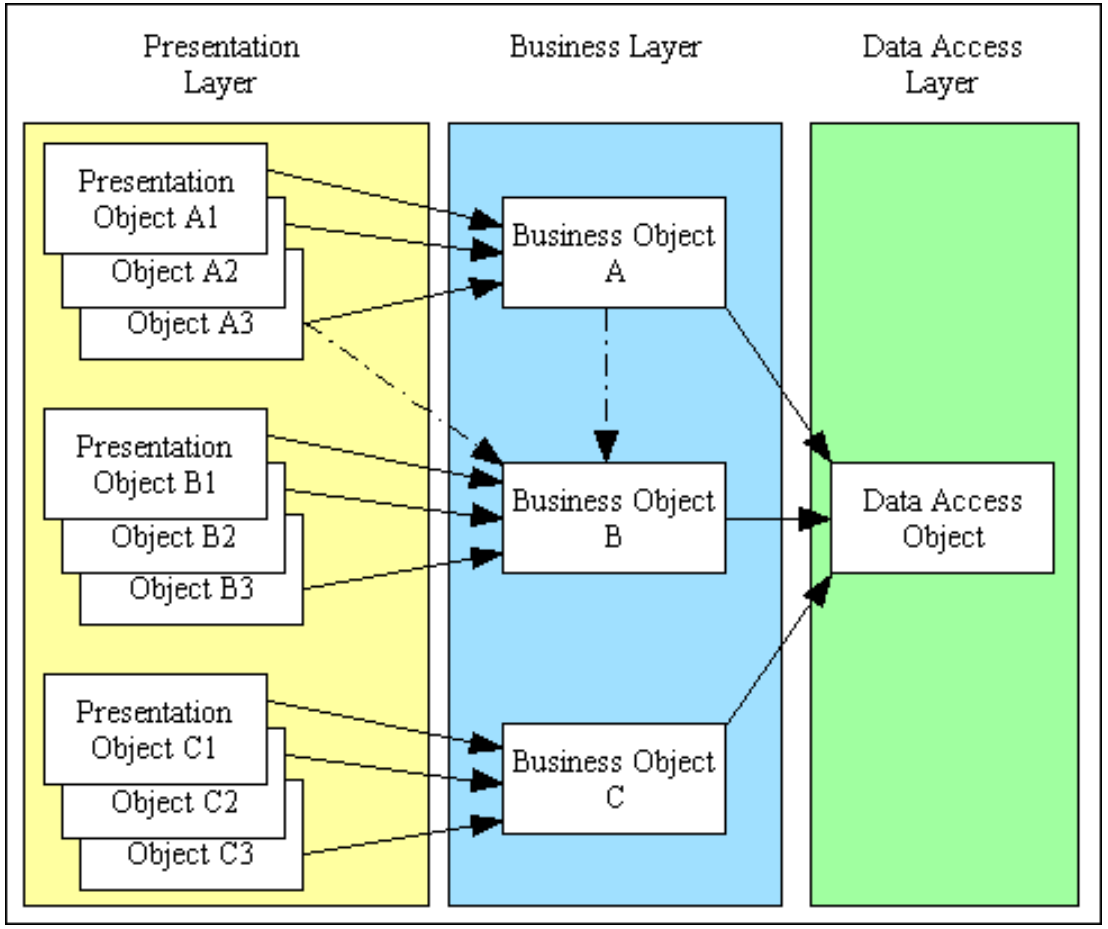
1. Based on class type:

```
com.firm.project.entity.Item
com.firm.project.entity.Customer
com.firm.project.repository.ItemRepository
com.firm.project.repository.CustomerRepository
com.firm.project.service.ItemService
com.firm.project.service.CustomerService
```

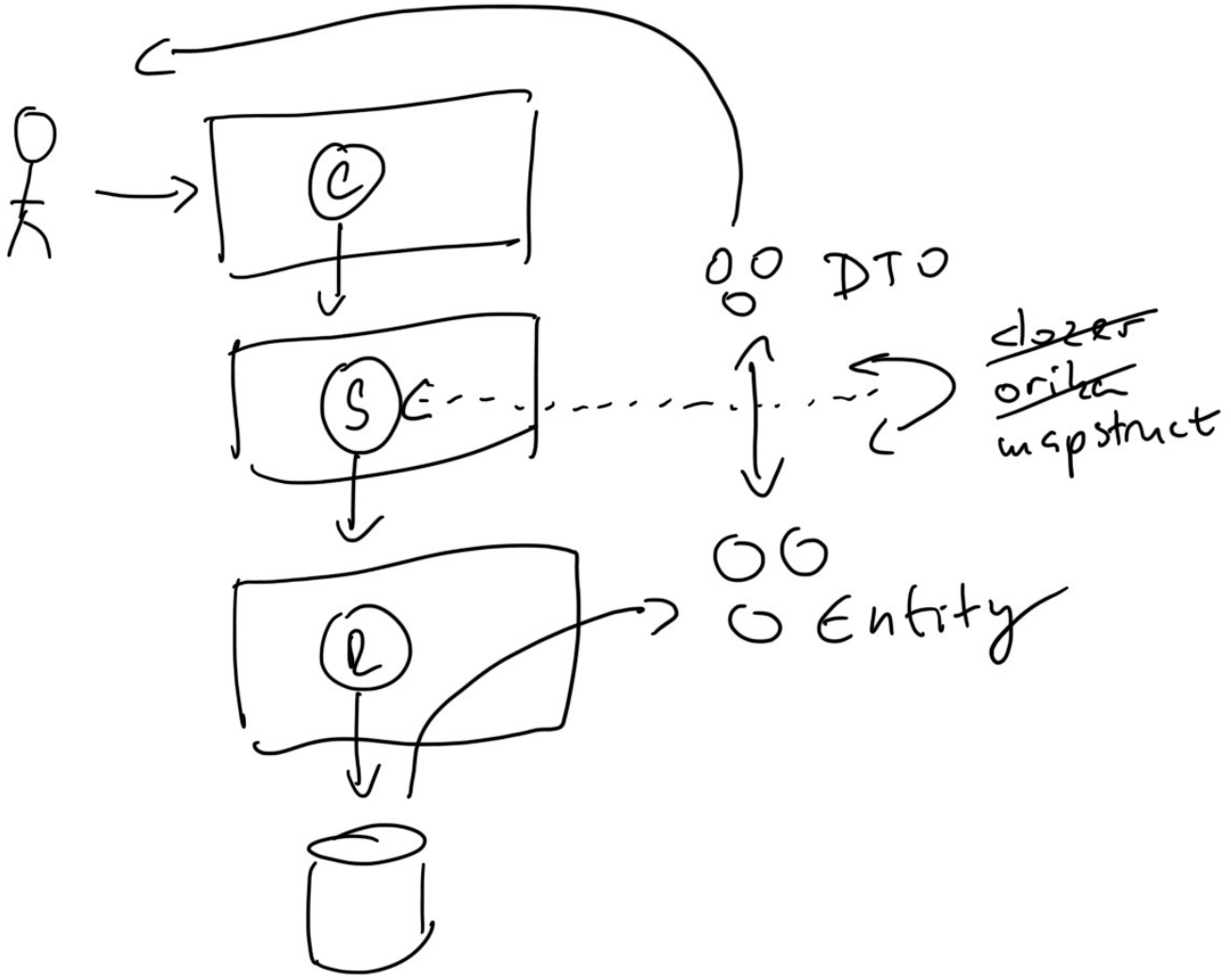
2. Based on use-case / entity:

```
com.firm.project.item.Item
com.firm.project.item.ItemRepository
com.firm.project.item.ItemService
com.firm.project.customer.Customer
com.firm.project.customer.CustomerRepository
com.firm.project.customer.CustomerService
```

Three Tier Architecture



Note: Output from Data Access Layer are entities, which are transformed in Service (Business) Layer by mapper (mapstruct) do DTOs, which are then returned by Presentation Layer to client.


C
S
R
DTO
~~dozer~~
~~orika~~
mapstruct
Entity

Java Bean, POJO, VO, DTO

- Java Bean:
  - Class, which has no-args constructor and access to attributes is via. getters / setters.
- POJO (Plain Old Java Object):
  - Simple Java object
- Value Object:
  - Object, which contains some „simple: value (for example Integer or Color.RED), VO is immutable.
- DTO (Data Transfer Object):
  - Object used for storing (and transfering) some data. DTOs are usually used when you read data from database using Hibernate (entities), but you want to return this data to client (DTOs)
- https://stackoverflow.com/questions/1612334/difference-between-dto-vo-pojo-javabeans

Singleton

- Pattern, Pattern, which restricts the instantiation of a class to one „single“ instance.
  - https://en.wikipedia.org/wiki/Singleton_pattern
- Is it pattern or anti-pattern?
  - IMHO it depends based on implementation:
    - If you were to use singleton in a way which is described in wikipedia, then it would be an anti-pattern.
    - But if you use for example IoC container to achieve the goal to have just one instance of some class, then it’s pattern.
  - When would you have issues if you were using Singletons? Especially in test classes.

Lazy initialization

- Lazy initialization isn’t design pattern, but it’s an important concept, when you delay initialization of some object until the first time it’s needed:
  - https://en.wikipedia.org/wiki/Lazy_initialization
- Sometimes using lazy initialization can be best practice, otherwise it’s exactly opposite.
- Usages:
  - Lazy relations in Hibernate entities

Immutable

- Immutable object (object, whose state cannot be modified after it's creation – for example String) is thread-safe. But make sure that object is really immutable!!!
  - https://dzone.com/articles/do-immutability-really-means
- Designing immutable object isn't always easy. For example first class isn't immutable, because you can change inner state from the outside:

```
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

- There are many classes, whose operations are thread-safe. It's then possible to have just single instance of such class in the whole multi threaded application.
- Examples of thread-safe objects:
  - DataSource
  - RestTemplate, JdbcTemplate
  - JAXBContext
  - PrettyTime
    - https://www.ocpsoft.org/prettytime/
    - Beware that in PrettyTime constructor is logic which loads i18n messages to memory (this isn't best practice and you shouldn't design your classes this way!!!). But because PrettyTime class is thread-safe, it's possible to have just single instance of this class in the whole application
- https://www.baeldung.com/java-thread-safety

Thread safety

- In multi-threaded applications it’s a good idea to use classes, whose operations are atomic / thread safe:
  - AtomicInteger, synchronized collections, StringBuffer, …
- If you don’t find appropriate class, use ReentrantLock (Lock)
- Don’t use “synchronized” keyword

Callback, Future, Promise, Reactive programming

- In some situations (especially in multi-threaded programming) it may be useful to call some method (A) and specify callback (B), which will be called after method (A) completes it’s logic:
  - https://en.wikipedia.org/wiki/Callback_%28computer_programming%29
- Or you can return Future from method:
  - https://geekyrui.blogspot.com/2012/06/future-vs-callback.html
- Or Promise (in Java it’s called 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 is a design pattern, where one object (subject) contains list of its dependents (observers) and notifies them automatically of any state changes.
  - https://en.wikipedia.org/wiki/Observer_pattern
- EventBus is practically implementation of Observer pattern. There are several used implementations in Java:
  - Guava EventBus
  - greenrobot:eventbus (used in Android apps)

Messaging systems

- In case you need publish-subscribe communication between servers, look at:
  - RabbitMQ
  - Kafka

Observer, RxJava, Project Reactor

- Also RsJava is an implementation of Observer pattern:
  - https://github.com/ReactiveX/RxJava

Builder

- Design pattern is designed to abstract and simplify creation of objects. Usually we could use constructor to simplify object creation. But what if we don’t need to set up all fields? Maybe sometimes we need to set up some fields and other times we just need to set up other fields. We could use multiple constructors, but their number would soon be unacceptable. In such scenario it’s a good idea to use Builder:
  - https://en.wikipedia.org/wiki/Builder_pattern
  - https://projectlombok.org/features/Builder

Factory method

- Factory method creates objects without having to specify the exact class of the object which will be created:
  - https://en.wikipedia.org/wiki/Factory_method_pattern
  - This design pattern is used quite a lot. Examples: Integer.valueOf(), List.of(), Path.of()
  - Sometimes factory methods use pool of objects (when they return immutable objects). Example: Integer.valueOf()

Strategy

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

Iterator

- Iterator is a design pattern, in which an iterator is used to traverse a container and access container’s elements so that this logic is decoupled from a container implementation:
  - https://en.wikipedia.org/wiki/Iterator_pattern

Service Locator

- Service Locator is a design pattern, which uses a central registry („service locator“), which on request returns some object:
  - https://en.wikipedia.org/wiki/Service_locator_pattern
  - This isn’t something you would want to implement and usually is replaced with Dependency Injection:
    - https://stackoverflow.com/questions/22795459/is-servicelocator-an-anti-pattern
    - https://stackoverflow.com/questions/1557781/whats-the-difference-between-the-dependency-injection-and-service-locator-patte

Dependency Injection & IoC

- Dependency Injection is at the heart of every modern container (like Spring or CDI). In this design pattern Injector creates objects and sets up their dependencies:
  - https://en.wikipedia.org/wiki/Dependency_injection
- Inversion of Control (IoC) is an architecture, where some framework creates objects and sets up their dependencies. One of implementation techniques of IoC is DI:
  - https://en.wikipedia.org/wiki/Inversion_of_control#Implementation_techniques

Spring

- If you’re starting with Spring in the age of Spring Boot, I would highly recommend this blog post to understand basics of Spring Framework:
  - https://www.marcobehler.com/guides/spring-framework

Request / Response mapper

- Especially when you use Hibernate and you send objects to client in JSON / XML format, you will use Request / Response mapper pattern.
- Basically it’s all about classes, which transform some POJO to another POJO (for example entity -> DTO).
- http://www.servicedesignpatterns.com/RequestAndResponseManagement
- MapStruct generates implementation of this pattern:
  - http://mapstruct.org/

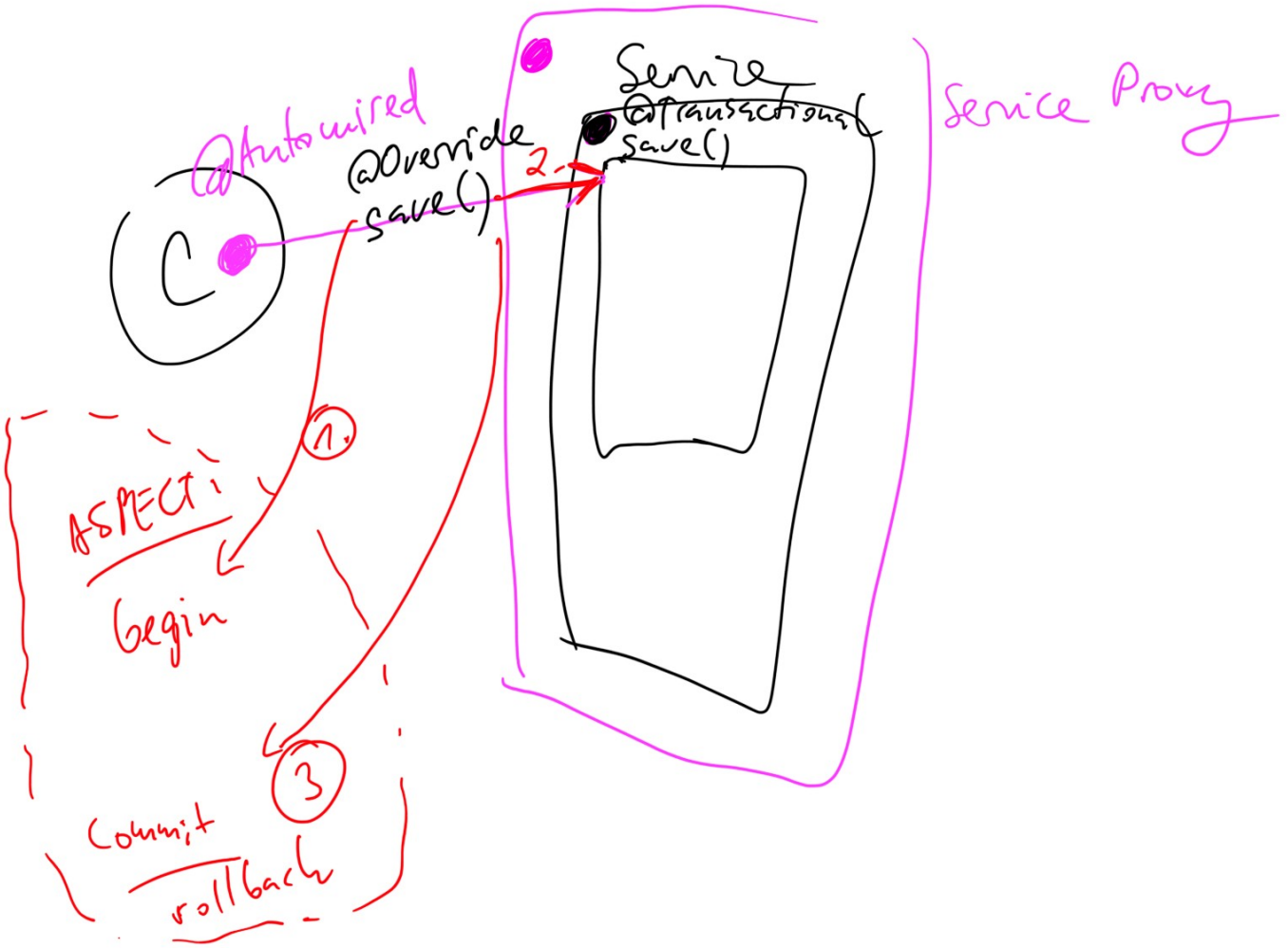
Scope

- Spring implements following scopes:
  - Singleton, prototype, request, session (+ it’s possible to create your own scope)
    - http://www.baeldung.com/spring-bean-scopes
- CDI has following scopes:
  - Request, session, application, dependent, conversation
    - http://docs.oracle.com/javaee/6/tutorial/doc/gjbbk.html

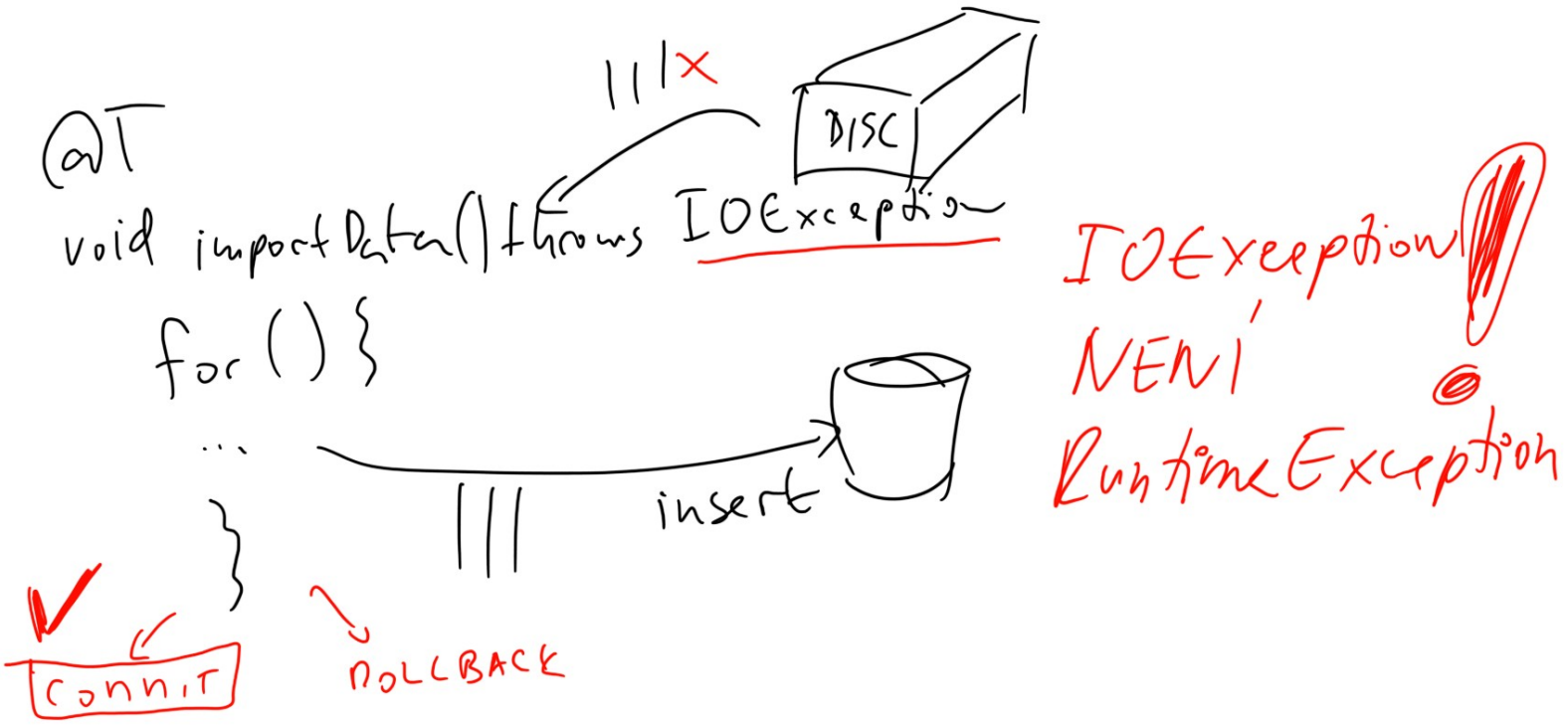
Proxy

- Proxy is a class, functioning as an interface to something else:
  - https://en.wikipedia.org/wiki/Proxy_pattern
  - Spring AOP creates proxy objects, which call logic, specified by annotations like: @Transactional, @Cacheable, @Async, @PreAuthorize.
  - Hibernate uses proxies for lazy relations.

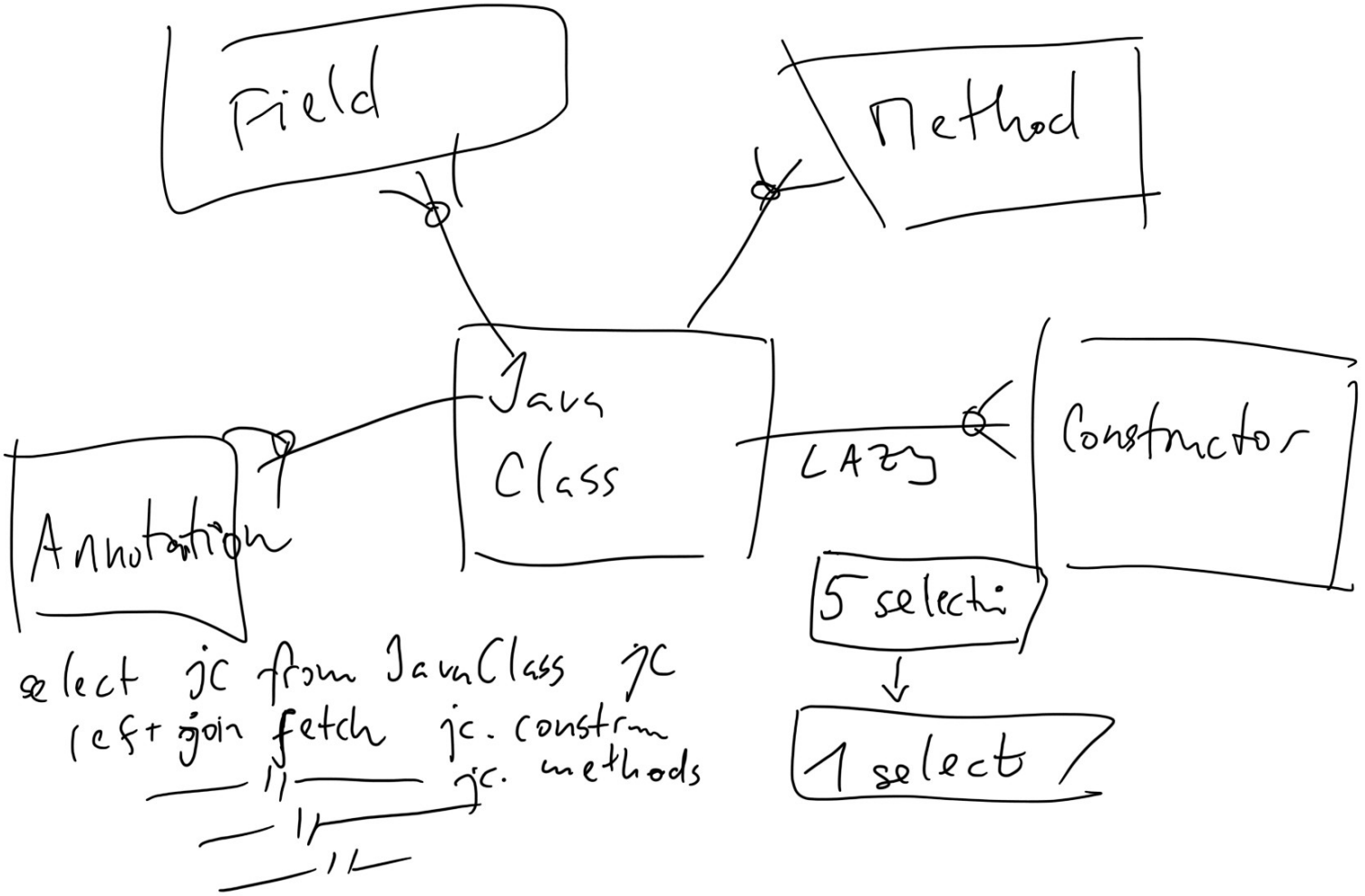
With Spring AOP make sure that you go through proxy!!!



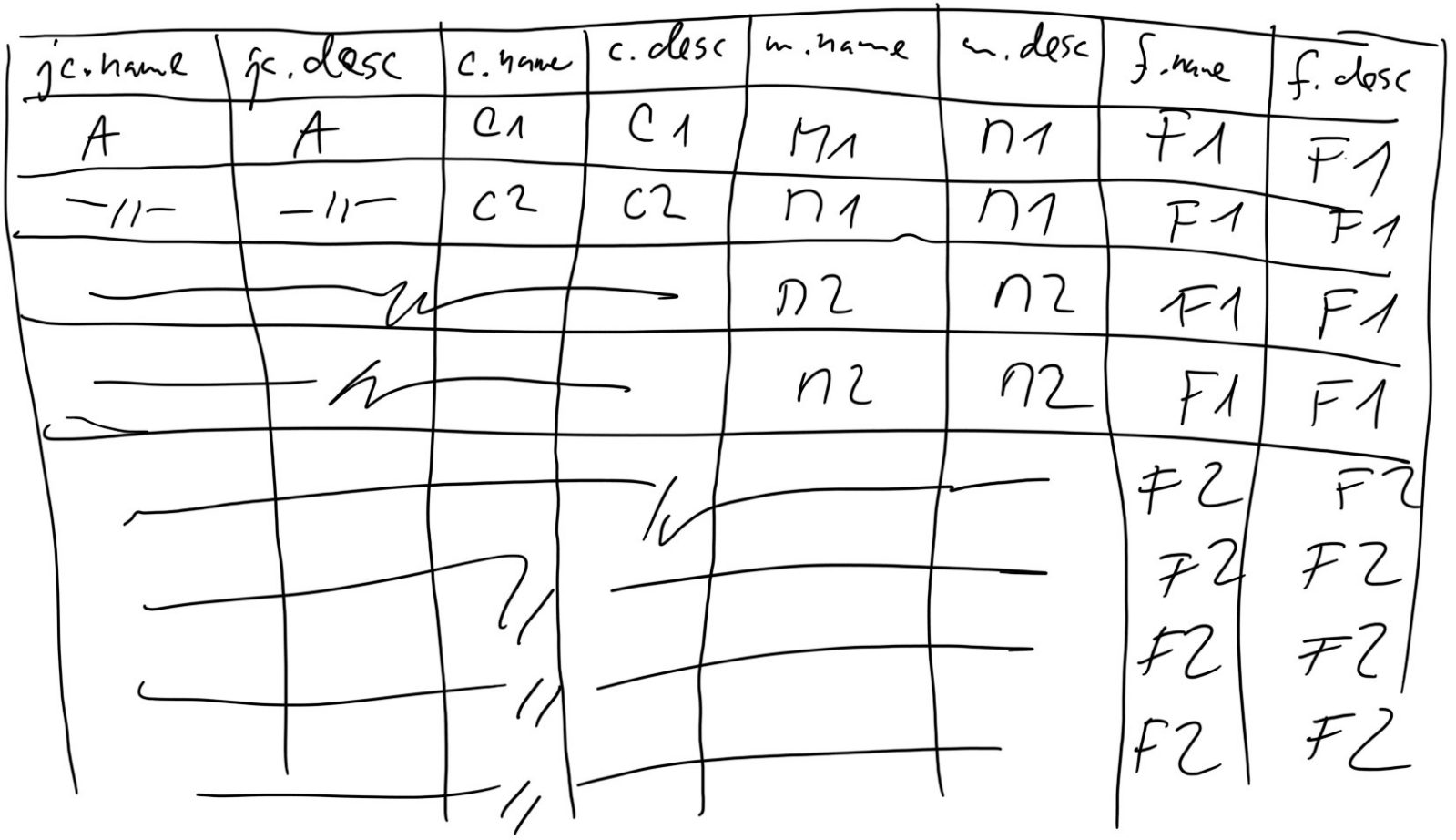
Warning! Out-of-the-box is transaction rolled back only when you throw RuntimeException!



Beware left join fetch (or entity graph, or eager relation) on a relation to a collection of entities!!! 1/2



It will work, but … in ResultSet will be lots of duplicities! 2/2



jc.name	jc.desc	c.name	c.desc	m.name	m.desc	f.name	f.desc
A	A	c1	c1	m1	m1	f1	f1
-"-	-"-	c2	c2	m1	m1	f1	f1
				m2	m2	f1	f1
				m2	m2	f1	f1
						f2	f2
						f2	f2
						f2	f2
						f2	f2

AOP

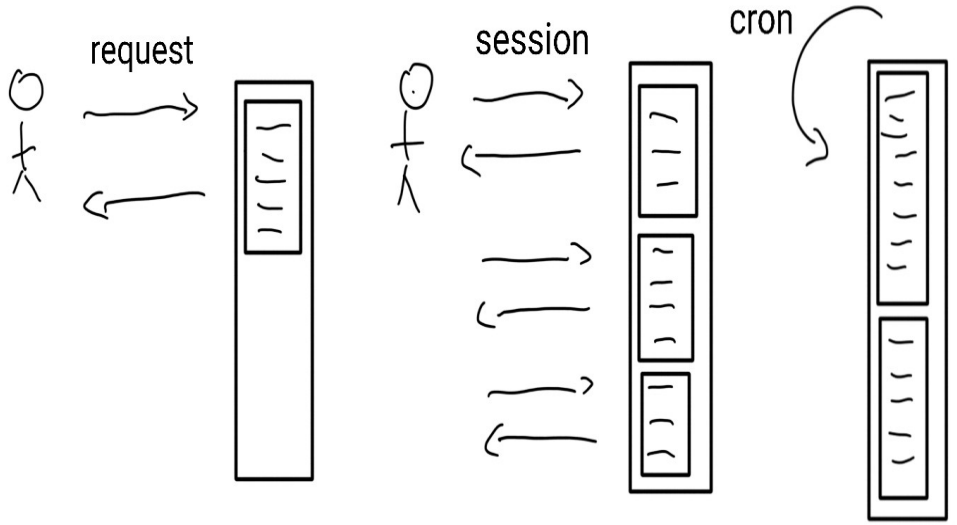
- Using AOP you can separate „cross-cutting concerns“ to aspects and declaratively set it up on some method, so that this logic is called before, after or around (before & after) some method.
  - Typical usage in Spring: @Transactional, @Cacheable, @Async, @PreAuthorize

Interceptor

- Interceptor automatically and transparently intercepts input request or output response, so it’s basically very similiar to AOP (just implemented differently):
  - https://en.wikipedia.org/wiki/Interceptor_pattern
- Typical use-case: javax.servlet.Filter
  - Spring Security is based on Filter
  - MDC for requests can be also set up using Filter (next slide)

MDC (Mapped Diagnostic Context)

- MDC (Mapped Diagnostic Context) is used in logging:
  - https://dzone.com/articles/mdc-better-way-of-logging-1
  - https://spring.io/projects/spring-cloud-sleuth



Decorator

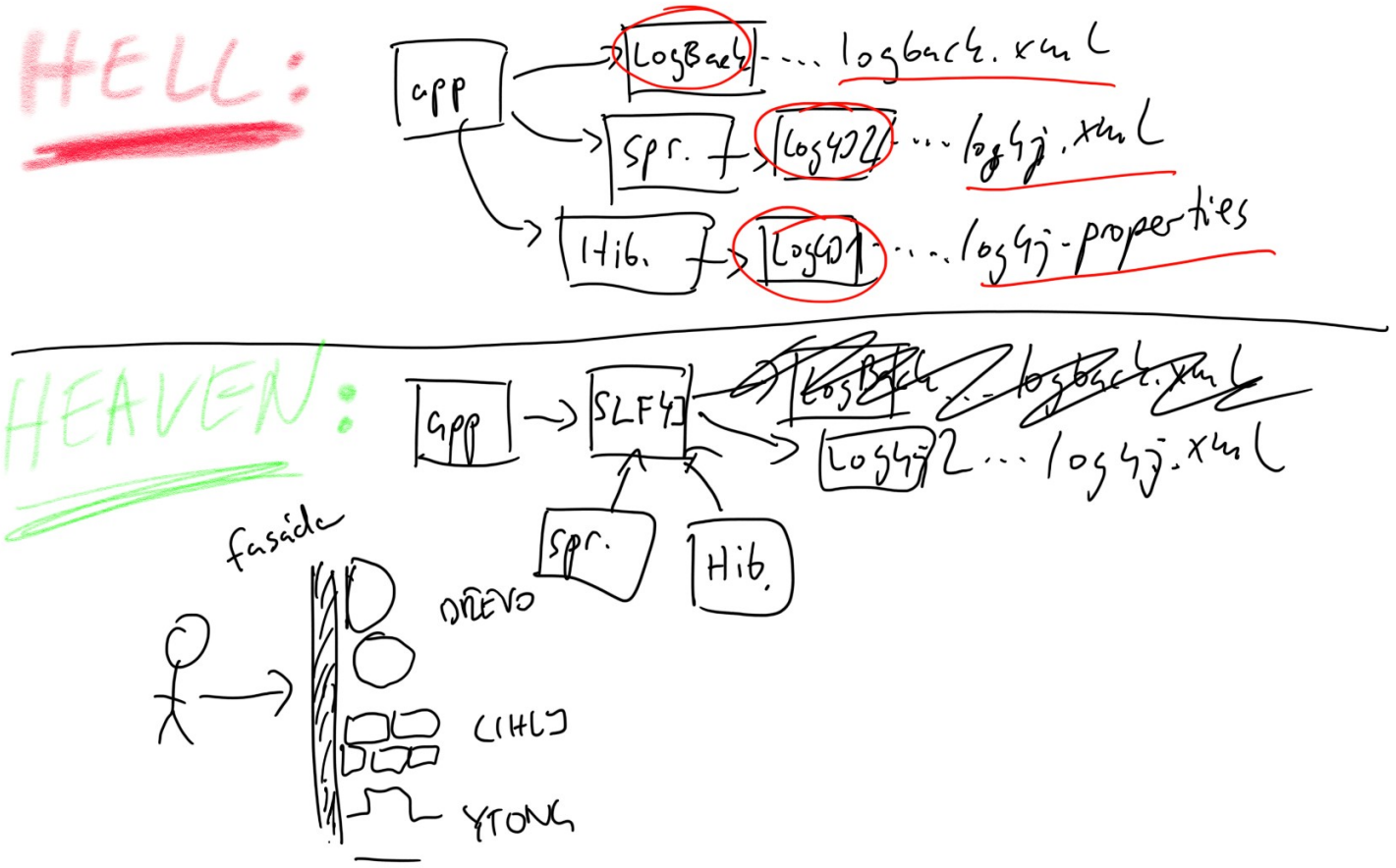
- Decorator pattern is a design pattern that allows behavior to be added to an individual object, dynamically, without affecting the behavior of other objects from the same class:
  - https://en.wikipedia.org/wiki/Decorator_pattern
- Difference between Decorator & Proxy (Proxy adds functionality at compile time, Decorator adds functionality at runtime):
  - https://stackoverflow.com/a/25195848/894643

Adapter

- Adapter allows the interface of an existing class to be used as another interface:
  - https://en.wikipedia.org/wiki/Adapter_pattern
  - https://refactoring.guru/design-patterns/adapter

Facade

- Facade shields one part of system from another. Most used Facade: SLF4J :-)
  - https://en.wikipedia.org/wiki/Facade_pattern



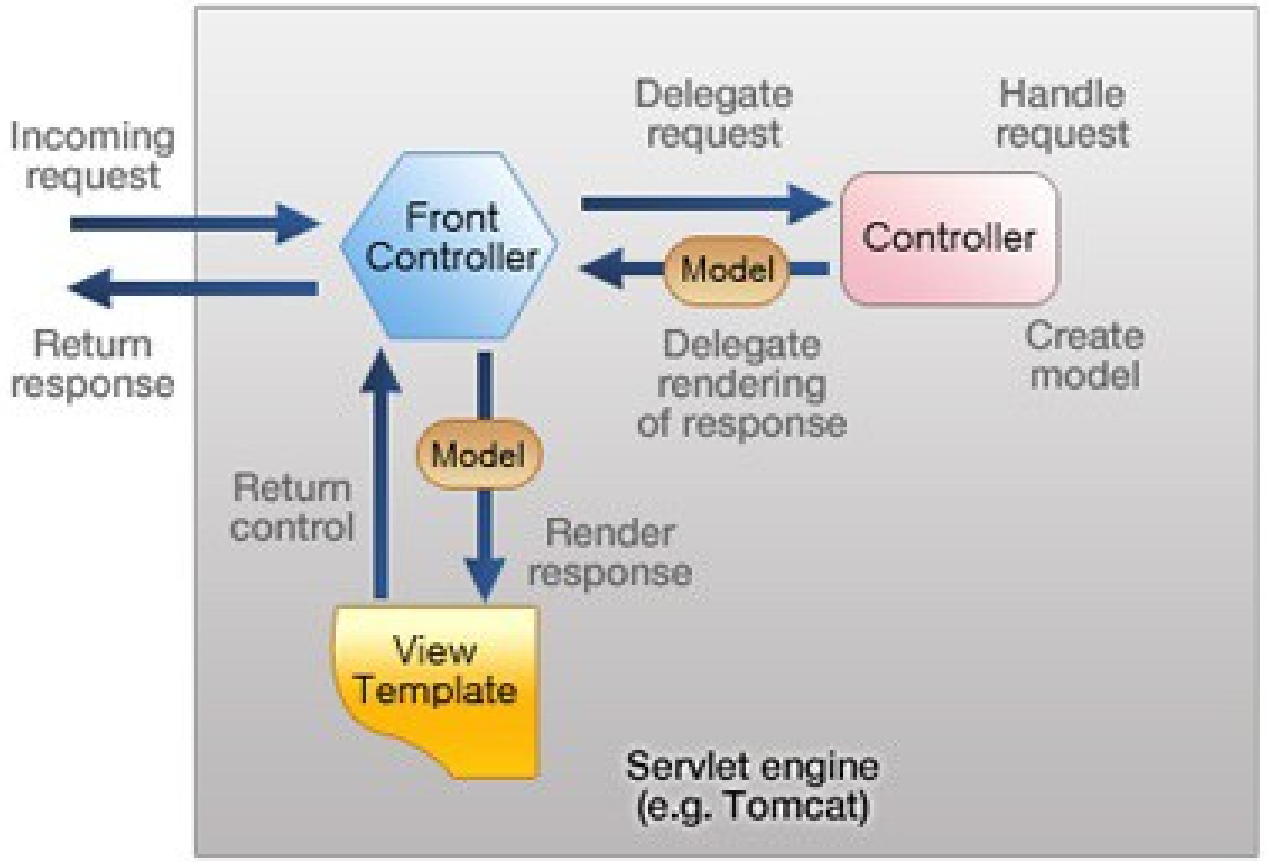
Object pool

- In various situations we need to pool some objects:
  - Threads:
    - http://tutorials.jenkov.com/java-util-concurrent/executorservice.html
  - Connection to database:
    - https://github.com/brettwooldridge/HikariCP
  - Integers:
    - Integer.valueOf()
  - Beware! You should create your own object pool only when you know what you’re doing! :-)
    - https://softwareengineering.stackexchange.com/questions/115163/is-object-pooling-a-deprecated-technique

Open In View (Anti)pattern

- Beware! This mechanism is automatically turned on in Spring Boot!!! DO NOT USE!!!
  - https://vladmihalcea.com/the-open-session-in-view-anti-pattern/

Front Controller & MVC



Note: MVC on server-side nowadays isn't used very often,
but Front Controller still makes sense and in Spring Boot
the DispatcherServlet is an implementation of Front Controller.

REST

- REST is an architectural style for providing standards between computer system on the web, making it easier for systems to communicate with each other.
  - https://en.wikipedia.org/wiki/Representational_state_transfer
  - Basic of REST is two types of URL: first represents list of items, second item's detail and GET, POST, PUT, DELETE operations (+ PATCH operation).
- PUT vs. PATCH:
  - https://stackoverflow.com/questions/28459418/rest-api-put-vs-patch-with-real-life-examples
- GET operations should be idempotent!!!
- For REST API documentation is OpenAPI:
  - https://springdoc.org/

HATEOAS

- REST is super flexible, but sometimes too much flexible … in such cases it might be useful to use HATEOAS. What is it? It’s a subset of REST and the basic premise is, that in the result from server are not just data, but also other meta-information:
  - https://en.wikipedia.org/wiki/HATEOAS
  - https://spring.io/projects/spring-hateoas

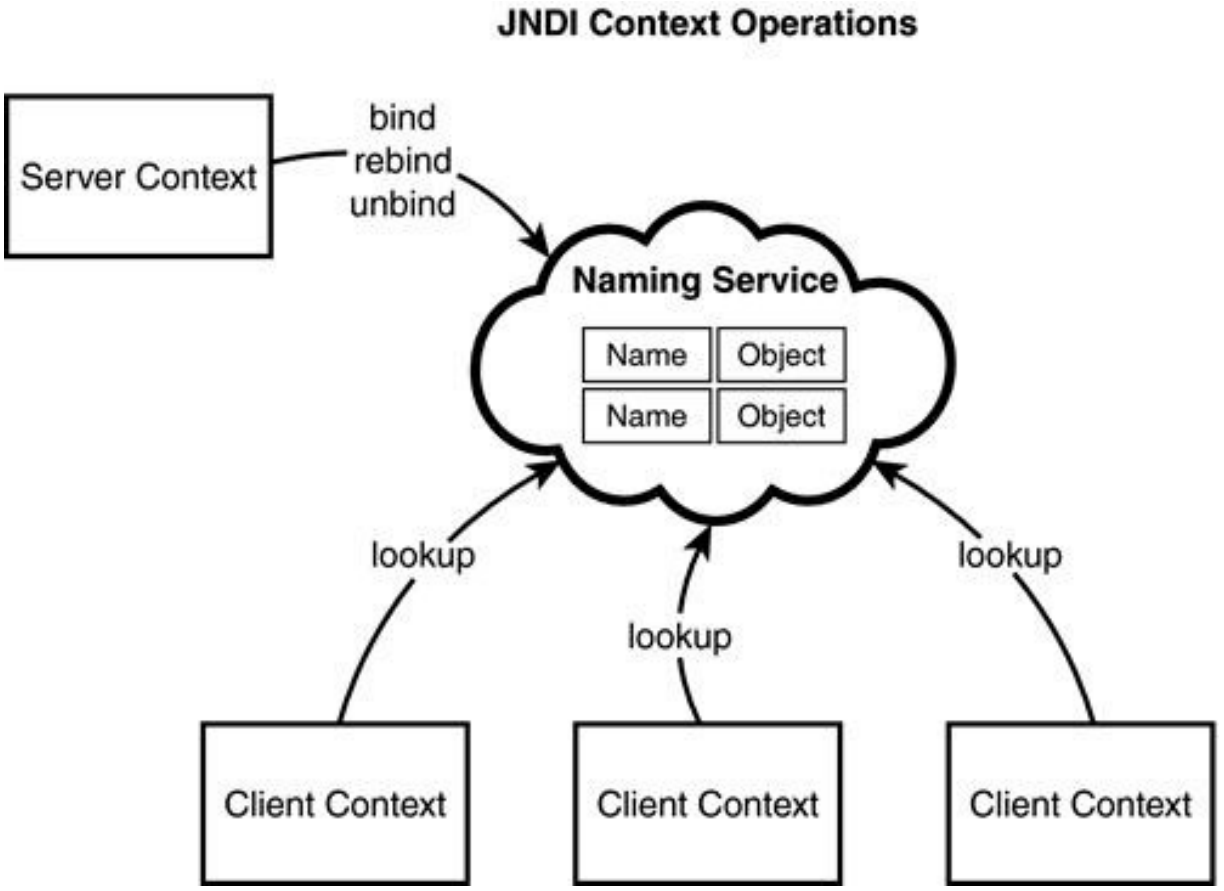
HAL

- HAL (Hypertext Application Language) is HATEOAS implementation:
  - https://en.wikipedia.org/wiki/Hypertext_Application_Language
  - https://stackoverflow.com/questions/25819477/relationship-and-difference-between-hal-and-hateoas
- Spring Data REST is built on HAL:
  - https://spring.io/projects/spring-data-rest
  - It’s a great tool for REST API generation. It’s not super flexible, but it does its job perfectly.

GraphQL

- REST has a disadvantage that for every use-case you have standalone endpoint. If your number of endpoints might “explode”, then take a look at GraphQL:
  - https://graphql.org/
  - GraphQL is basically something like SQL for your 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

- Lots of classes implement AutoCloseable interface. Then instead of calling close() method, it’s better to use try-with-resources:
  - https://www.baeldung.com/java-try-with-resources
- Note:
  - In Java 9 try-with-resources was enhanced:
    - https://dzone.com/articles/try-with-resources-enhancement-in-java-9

Important tools

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

Functional Programming

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

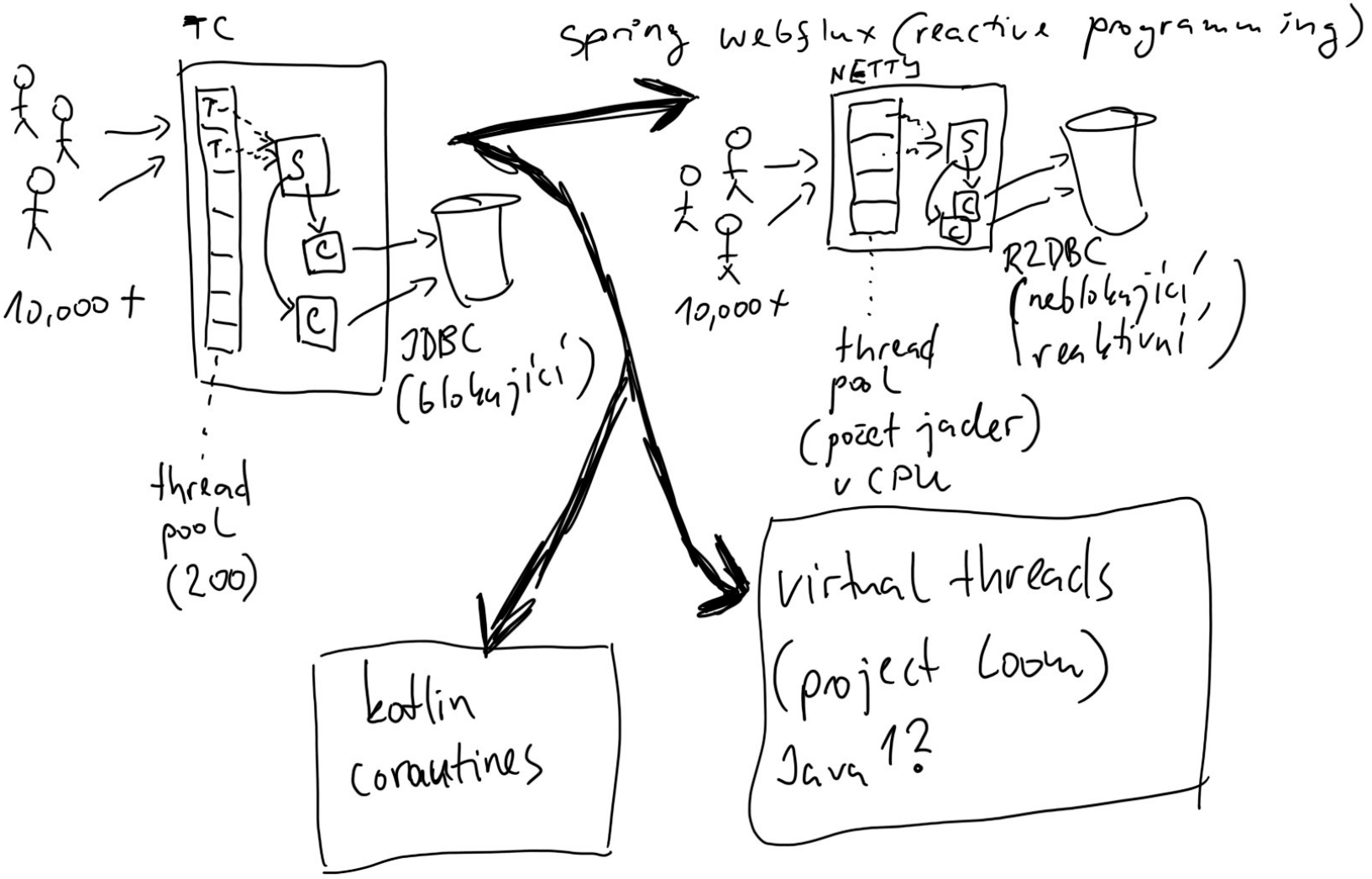
Best Practices I.

- Threads:
  - Don’t use new Thread(), use Executor (thread pool)
- In constructors (and getters / setters) shouldn’t be logic, that should be elsewhere!!!
- For other best practices I highly recommend these books:
  - Joshua Bloch: Effective Java
  - Vlad Mihalcea: High Performace Java Persistence
  - Chris Richardson: Microservices Patterns
- Modern best practices for testing in Java:
  - https://phauer.com/2019/modern-best-practices-testing-java/

Lombok etc.

- For boilerplate generation is usually used Lombok, but you should be careful and generate only that, which you need / want!!!
  - https://deinum.biz/2019-02-13-Lombok-Data-Ojects-Arent-Entities/
- Another interesting library:
  - https://github.com/immutables/immutables
    - It’s for creation of Value Objects using builder pattern (great for DTOs)
  - Or AutoValue:
    - https://github.com/google/auto/tree/master/value
- From 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

- With Reactive programming is used Event loop princip:

Benefits of reactive programming (better throghput and constant usage of RAM):
https://www.quora.com/What-is-the-better-web-server-stack-Nginx-or-Apache

Various libraries for database access