# Microservices patterny

# Materialized view

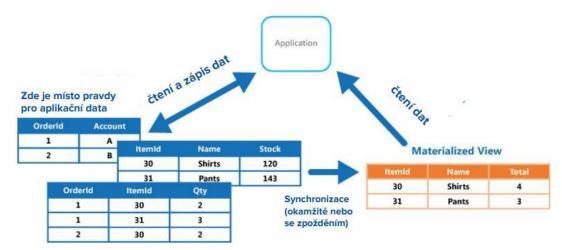
- single source of truth (místo pravdy dat)
  - o = jediné místo v systému (/ organizaci), kde jsou 100% aktuální data

#### Motivace:

- data čteme výrazně častěji než zapisujeme
- čtení dat je pomalejší než např. odezvy, které potřebujeme na UI
- dotaz čtoucí data je náročný na výkon

#### Řešení:

- z SSoT si dopředu vytvoříme pohled na data, který obsahuje pouze data které potřebujeme
- data jsou předzpracována do modelu, ve kterém potřebujeme data číst
- tento pohled používáme pouze na čtení
- do tohoto pohledu nikdy nezapisujeme přímo
  - o měníme ho až po změně v SSoT
  - o buď současně se zápisem do SSoT, nebo se zpožděním (asynchronně nebo v dávce)



# **Event sourcing**

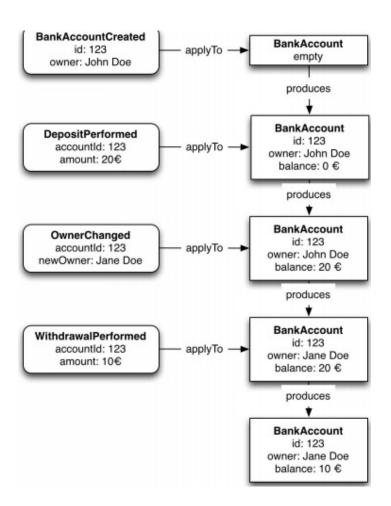
- klasický přístup při změně dat:
  - o první aplikace přímo update stav druhé aplikacve



- event sourcing je pattern, při kterém:
  - o veškeré změny do stavu aplikace jsou ukládány jako eventy
  - o eventy jsou uloženy v sekvenci ve které byly provedeny
  - o mezi updatující aplikací a updatovanou aplikací je mezivrstva pro uložení eventů (DB,



# Sekvence eventů >>> BankAccountCreated id: 123 owner: John Doe DepositPerformed accountld: 123 amount: 20€ DepositPerformed accountld: 123 newOwner: Jane Doe WithdrawalPerformed accountld: 123 amount: 10€



#### - výhody:

- o performance aplikace zapisující změny není blokována apl. do níž se změny zapisují
- o **rekonstrukce stavu aplikace** pokud bychom přišli o stav aplikace, jsme schopni jej zreplikovat od každého okamžiku novou aplikací eventů
- o **vrácení se k jakémukoliv minulému stavu aplikace** vrátíme se jednoduše tím, že znovu aplikujeme event až do požadovaného okamžiku
- rollback zjistíme-li, že chceme poslední update zrušit, tak provedeme inverzní operace k eventu

#### pozor na:

- side efekty např. poslání klientovi 2x stejný email řešením je oddělení side efektů od stavových změn
- o úložiště eventů se nehodí na generování reportů a dotazování na data

#### Event vs. Command

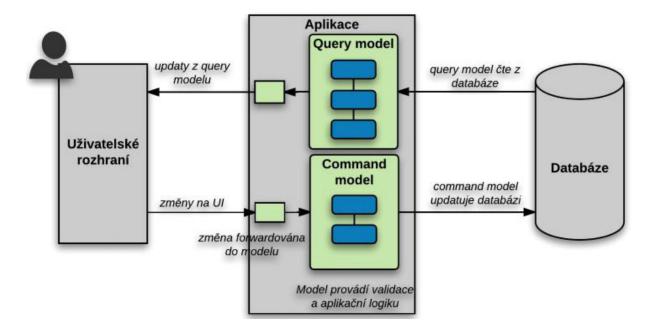
- event = co se stalo [např. Order Placed]
- **command** = co se má udělat [např. Retrieve payment]

# CQRS (Command Query Responsibility Segregation)

- odděluje model pro zápis od modelu pro čtení
- operace by měla buď změnit stav objektu, anebo vrátit výsledek (ne obojí najednou)
- vznikají dva typy operací:
  - o Commands mění stav objektu nebo celého systému (tzv. mutátory)
  - Queries vrací výsledek, nemění stav objekt

#### - důvody:

- o R/W v jednom modelu vede u složité aplikace ke komplikovanému modelu/dotazům
- zcela odlišné nefunkční požadavky na čtení a zápis navzájem se blokují, prodlužují odezvy



# Microservice

- MS architektura redukuje komplexitu pomocí dekompozice systému na moduly
  - o moduly jsou dobře ohraničené z hlediska fungování a účelu
- microservice by měla realizovat jednu **business capabilitu** (schopnost/dovednost)
- nemá nic společného s velikostí, je o maximálním respektování Single Resp. Principle
- fungují zcela samostatně a autonomně
- preferuje, aby měla vlastní DB, do které napřímo nepřistupuje žádný další systém (/MS)
- dvě MS mohou sdílet data a odpovědnost (přes třetí subjekt)
  - o závislost je však co nejmenší a nejvolnější

# DDD - Domain Driven Design

- metodika SW vývoje, která se soustředí na správnou dekompozici problému do domén a realizaci takto dekomponovaného systému

Bounded Context – zasazuje část modelu do kontextu, kde má svůj unikátní význam, chování a izolaci

**Aggregate** – objektový graf, který spravujeme jako celek

Entity – business object reprezentující doménový koncept se stabilní identitou

Value object – immutable object reprezentující kompozitní hodnotu objektu

Domain service – domain behavior, které se neperzistuje (např. vypočti daň)

Domain event – objekt reprezentující změnu v doméně

Repository – objekt zodpovědný za perzistenci agregátu

# Rozdělení problému na domény

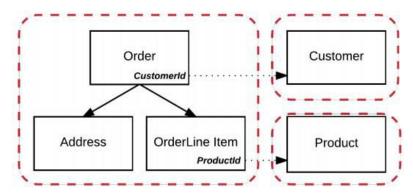
- Jak identifikovat klíčové domény řešeného problému?
- Jak se zaměřit na **oblasti**, které jsou **důležité pro business**?
- Jak vypadají **hranice** mezi doménami?
- Jak přiřadit lidské zdroje k doménám?
- měl bych mít možnost kdykoliv vzít jednu oblast a s minimálním úsilím/dopadem (minimální rework, testování,...) do ostatních domén:
  - o nasadit opravu
  - upravit funkcionalitu
  - o migrovat funkcionalitu na jinou technologii
  - o rozšířit z pohledu CPU a datového úložiště
  - o outsourcovat do jiného oddělení nebo společnosti

#### - Core Domain

- o zpravidla reprezentuje core business firmy (nejvíce peněz, kompetitivní výhoda,...)
- o dáváme na ni nejlepší vývojáře
- o pohled spíš jako na business produkt než jako na projekt
- Generic Domain zpravidla vrstvy a tooly, které jsou využívány Core businessem
- **Supporting Domain** (ostatní)
- příklad (aukce):
  - Membership registrace, preference, detaily členů [= supporting]
  - Seller veškeré aktivity prodejce [= core]
  - Auction řízení a časování aukcí, práce s nabídkami [= core]
  - Listings seznamy položek pro dražbu [= supporting]
  - Dispute resolution řešení sporů mezi prodejci a kupci [= generic]

#### Agregát

- graf objektů (/tabulek), ke kterým se mohu chovat jako k uzavřené jednotce
- má vždy jeden root objekt a sadu dalších navázaných objektů
- linkuje se na další agregáty pomocí ld jejich root objektů (viz Ghost LL)
- jeden agregát = jeden command, který s ním pracuje
- scope transakce = agregát



#### **Bounded Context**

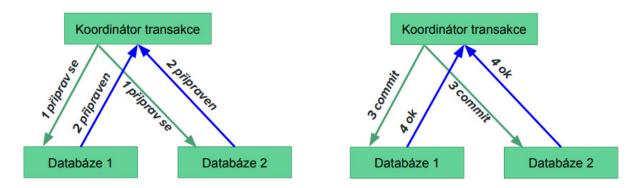
- zasazuje část modelu do kontextu, kde má svůj unikátní význam a chování
- BC = jedna microservice
- je o kontrole hranice, důsledném zapouzdření a izolaci
- AntiCorruption Layer zařizuje "čistou a jasnou" hranici mezi doménami
  - o veškerá komunikace a transformace dat mezi doménami probíhá přes tuto vrstvu

# Microservices – transakce apod.

- transakce musí splňovat ACIDitu
  - O Atomocity změny jsou buď realizovány jako celek nebo vůbec
  - o Consistency DB je konzistentní před a po transakci
  - o **Isolation** během transakce jsou data izolována od dalších transakcí
  - o **Durability** úpravy provedené po transakci jsou natrvalo uloženy v DB

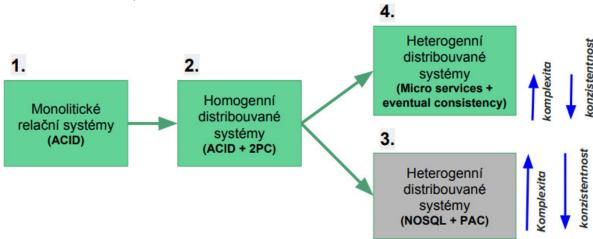
### Relační DB a dvoufázový commit (2PC)

- potvrdí-li všichni účastníci distribuované transakce připravenost realizovat operace, provádí se commit na všech těchto systémech, jinak se neprovede na žádném



- není vždy použitelný, protože:
  - o existující aplikace většinou **nepodporují transakce a 2PC** (není to XA resource)
  - o koordinátor tr. zavádí single point of failure (při jeho výpadku nejde zapsat nikam)
  - o protokol pro realizaci 2PC je velmi "upovídaný"
  - omezená propustnost (kvůli konzistenci musíme zamykat tabulky a serializovat operace)

# Geneze architektury

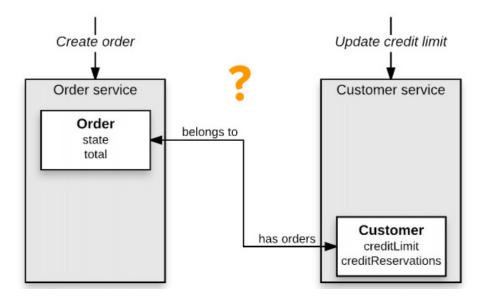


# Sága a Eventual consistency

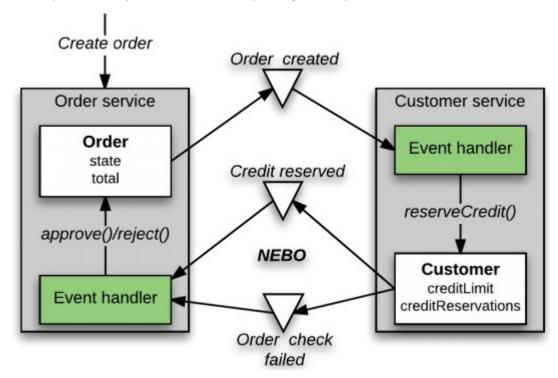
- **eventual consistency** = dočasně nekonzistentní systémy
- z technických či business důvodů nejsme schopni po celou dobu zaručit konzistentnost
   → navrhneme systém / business proces tak, aby byl použitelný i v nekonzistentních stavech
- systém je konzistentní jen v některých stavech, kterými prochází
- některé systémy se do plně konzistentního stavu nedostanou nikdy
- příklad:
  - hypotéka je konzistentní těsně před schválením
  - o v tu chvíli ke všem nemovitostem existuje výpis z katastru, ke všem osobám na hypotéce existuje doklad o příjmech apod.
  - o předtím byla hypotéka uložená, ale např. k některým osobám nebyl uložen příjem

#### Sága

- řeší konsensus mezi microservisami bez transakcí (pomocí správné dekompozice business procesu a eventual consistency)
- rozdělí komplexní BP na menší akce a protiakce (counteractions), které koordinuje a řídí pomocí zpráv a timeoutů
- **business process** = Saga, která obsahuje actions, counteractions, messages a timeouts
  - o operace v každém kroku BP má definovanou kompenzační operaci
- příklad:
  - zakládám objednávky a potřebuji zjistit, že suma otevřených objednávek není vyšší než kreditní limit zákazníka

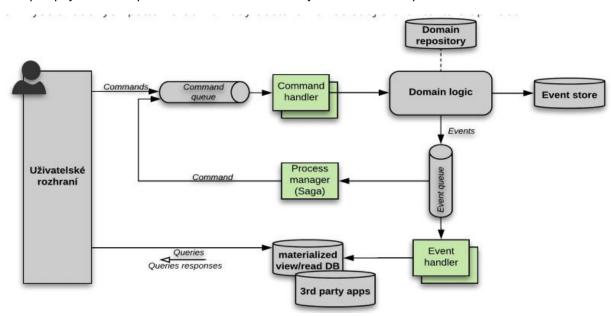


- o po založení objednávky je systém po nějakou dobu částečně nekonzistentní
  - mohl jsem založit objednávku, která překračuje kreditní limit zákazníka
- o řešení je možné, jelikož jsem upravil BP a částečně degradoval user experience
  - zákazník čeká na potvrzení objednávky
- obecně platí, že od jistého stavu systému (paretovsky optimální řešení) nemohu zlepšit některé jeho vlastnosti aniž bych degradoval jiné



# Materialized view + CQRS + Event sourcing + Saga

- po spojení těchto patternů dostáváme následující architekturu aplikace:



- **Command Queue** fronta, v níž se skladují commandy např. z UI, které jsou odebírány zaregistrovanými komponentami
- Command Handler(s) kód, který selektivně odebírá commandy a zpracovává je pomocí služeb doménové logiky
- **Domain Logic** služby nad DL, které provádí kód pro danou doménu (např. logika pro zpracování klienta banky, logika pro spotřební úvěr,...); při změnách generují eventy
- Event Store úložiště eventů
- Event Queue fronta eventů, které jsou odebírány zaregistrovanými komponentami
- **Event Handler(s)** kód, který selektivně odebírá eventy (ty jsou dále zpracovávány)
- Materialized View datové repository optimalizované pro čtení např. z Ul
- Process Manager / Saga implementace ságy, transformuje eventy do akcí nad DL