

10 - *Microservices patterny*

- Materialized view
- Event sourcing
- CQRS
- Eventual consistency
- Saga

Ing. David Kadleček, PhD

kadlecd@fel.cvut.cz, david.kadlecek@cz.ibm.com

Microservices Patterny

- Materialized view
- Event sourcing
- CQRS
- Eventual consistency
- Saga
- API Gateway (další přednáška)

Materializovaný pohled (Materialized view)

Místo pravdy dat (**single source of truth**) - jediné místo v systému (nebo organizace), kde jsou 100% aktuální data

Motivace

- Data čteme výrazně častěji než updatujeme.
- Čtení dat je pomalejší než např. odezvy, které potřebujeme na uživatelském rozhraní.
- Dotaz který čte data je náročný na výkon.

Řešení

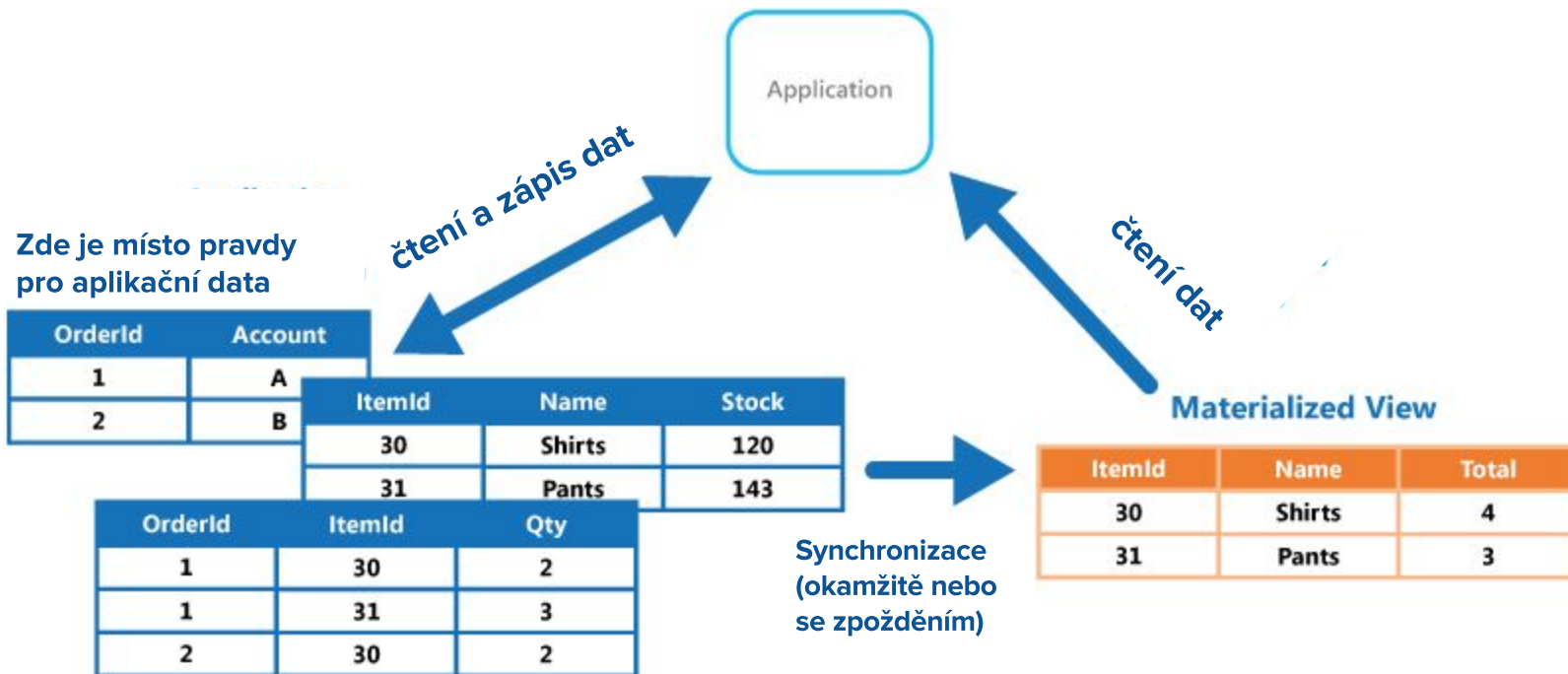
=> Z místa pravdy dat si dopředu vytvoříme pohled na data, který obsahuje pouze data které potřebujeme.

=> Data jsou předzpracována do modelu, v kterém potřebujeme data číst.

=> Tento pohled používáme pouze na čtení.

=> Do tohoto pohledu nikdy nezapisujeme přímo, měníme ho až po změně v hlavním místě pravdy - buď současně se zápisem do hlavního místa pravdy nebo se zpožděním (asynchronně nebo v dávce).

Materializovaný pohled (Materialized view)



Event sourcing

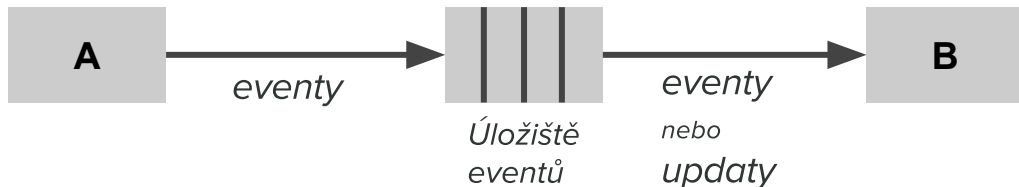
Klasický přístup při změně dat:

- první aplikace přímo updatuje stav druhé aplikace

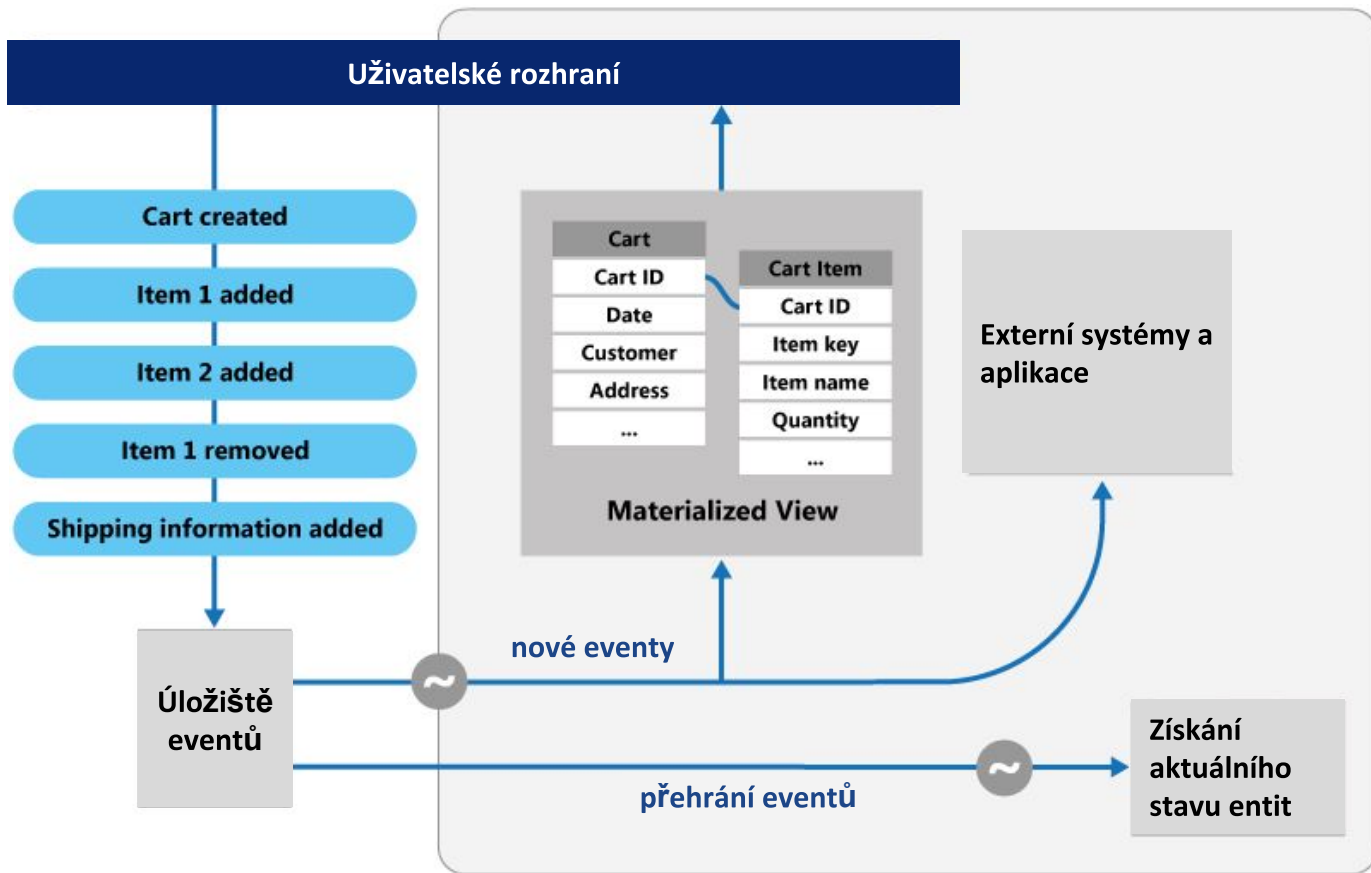


Event Sourcing je pattern při kterém:

- veškeré změny do stavu aplikace jsou ukládány jako eventy
- eventy jsou uloženy v sekvenci ve které byly provedeny
- mezi updatující aplikací a updatovanou aplikací je mezivrstva pro uložení eventů (databáze, fronta ...)

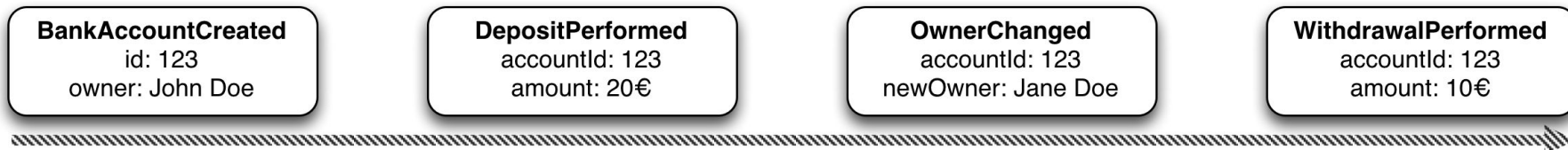


Event sourcing pattern

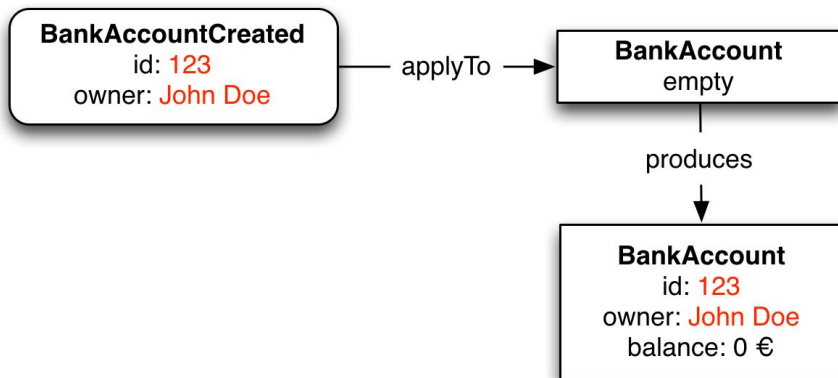


Event sourcing

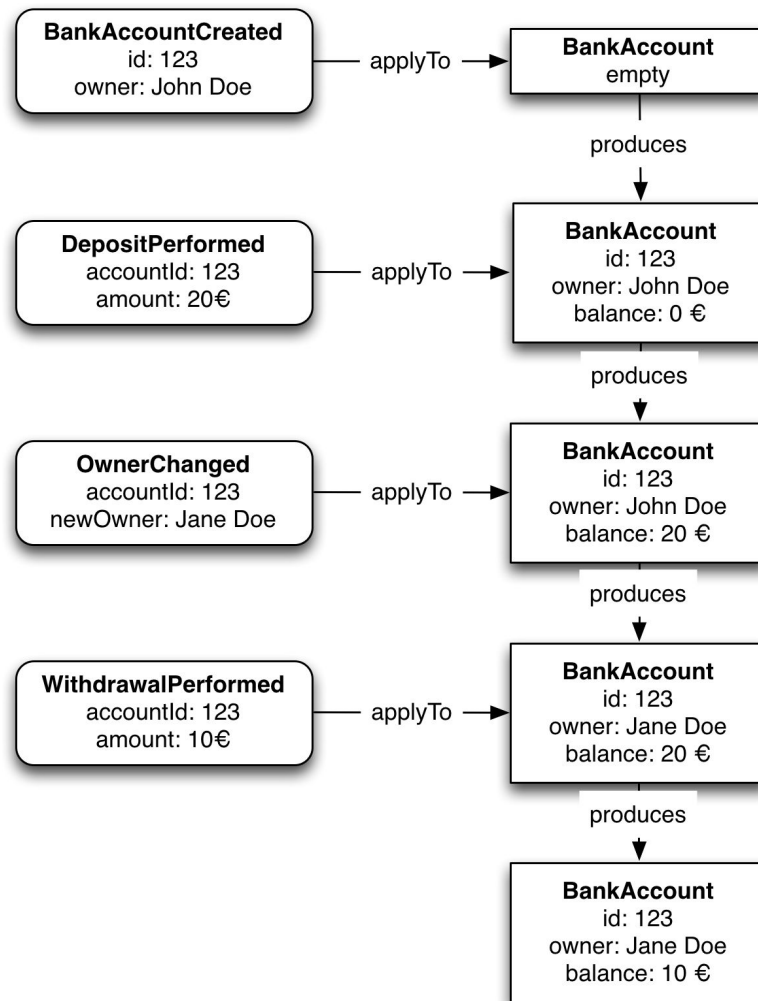
Sekvence eventů >>>



Aplikace eventů



Event sourcing



Event sourcing

Výhody:

- **Performance** - aplikace zapisující změny není blokována aplikací do které se změny zapisují
- **Rekonstrukce stavu aplikace** - když bychom přišli o stav aplikace, tak ho jsme schopni zreplikovat od každého okamžiku novou aplikací eventů
- **Vrácení se k jakémukoliv minulému stavu aplikace** - vrátíme se jednoduše tím, že znovu aplikujeme event až do požadovaného okamžiku
- **Rollback** - když zjistíme, že chceme poslední updatu zrušit, tak provedeme inverzní operace k eventům

Pozor na:

- **Side efekty** - např. abychom neposílali klientovi 2x ten samý email => řešením je např. oddělení side efektů od stavových změn
- Úložiště eventů se např. **nehodí na generování reportů a dotazování na data** => pokud toto chceme, tak eventy aplikujeme do úložiště pro čtení a čteme data až z něj

Event vs Command

Event - co se stalo



Command - co se má udělat



```
public class OrderPlacedEvent implements Event {
    UUID eventId;
    UUID srcId;
    List items;
    public OrderPlacedEvent(UUID SrcId, List<Item>
items){
        //set instance variables
    }
}
```

```
public class RetrievePaymentCommand implements Command{
    UUID commandId;
    String accountId;
    BigDecimal amount;
    public RetrievePaymentCommand(String accountId,
BigDecimal amount ){
        //set instance variables
    }
    public void execute(){
        //code to be executed
    }
}
```

CQRS

CQRS odděluje model pro zápis od modelu na čtení = **Command Query Responsibility Segregation**

Hlavní idea CQRS je:

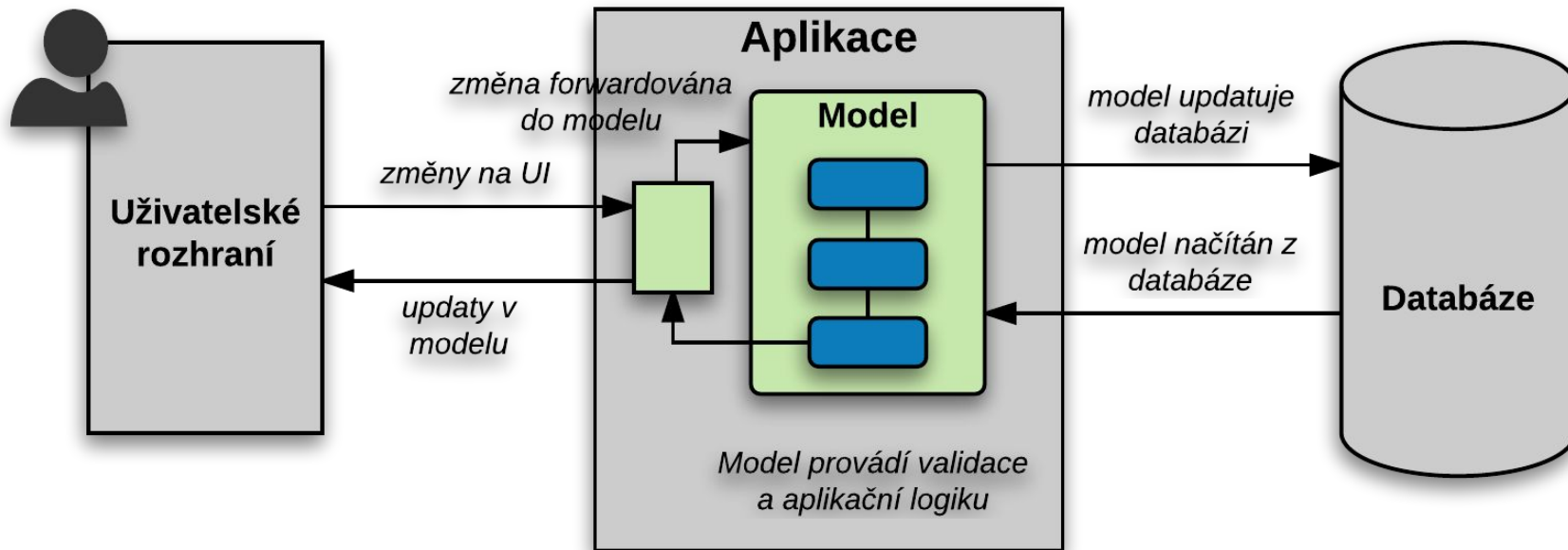
Operace by měla buď změnit stav objektu nebo vrátit výsledek, ale ne obojí najednou - *odpovídání na otázku nemění otázku*. Když provedeme takovou segregaci, tak budeme mít vždy dva typy operací:

- **Commands** - mění stav objektu nebo celého systému - tzv. *mutátory*
- **Queries** - vrací výsledek a nemění stav objektu

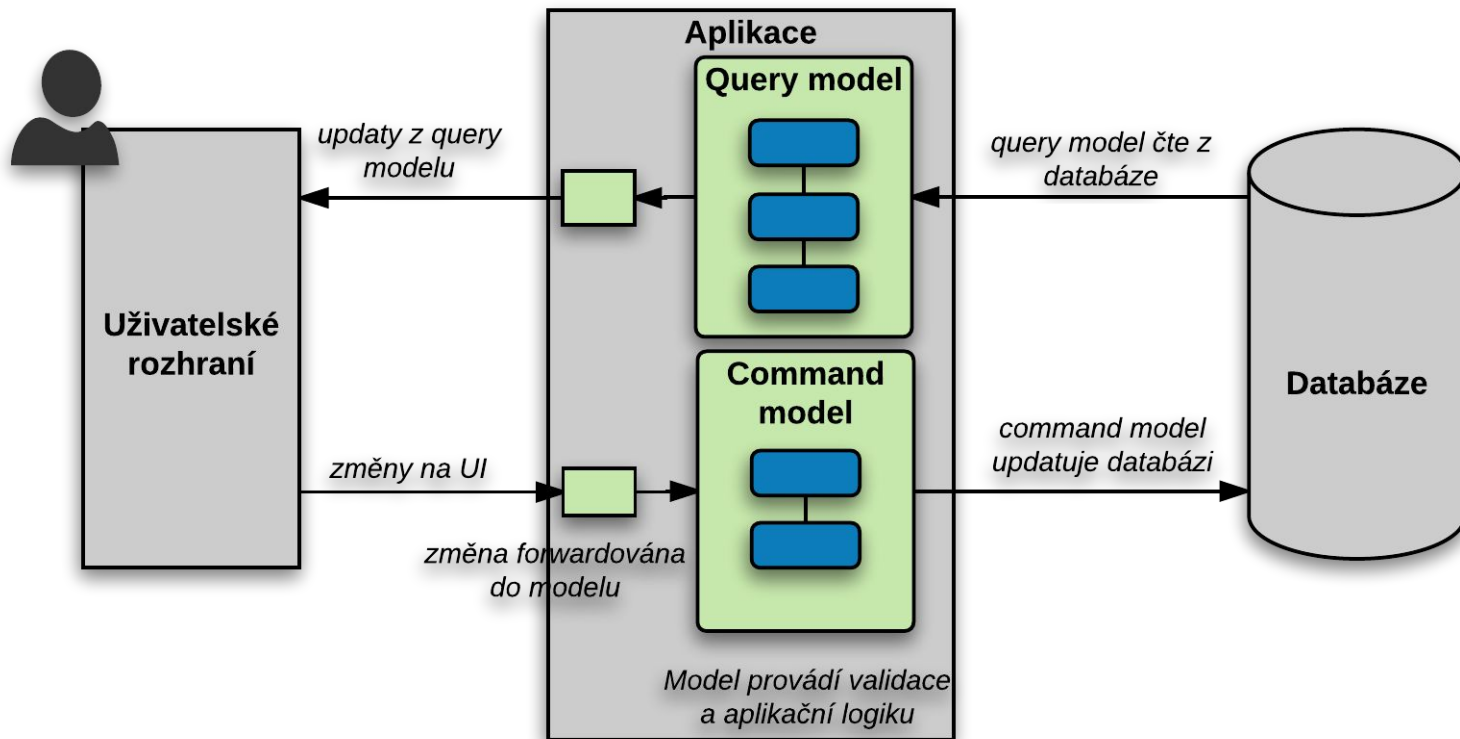
Důvody pro CQRS:

- U složité aplikace vede spojení požadavků pro zápis a čtení do jednoho modelu k příliš komplikovanému modelu a komplikovaným dotazům
- Aplikace má zcela odlišné nefunkční požadavky na čtení a zápis - zápis a čtení se navzájem blokují a prodlužují odezvy

Klasicky stavěná aplikace



CQRS - oddělení modelu pro čtení a zápis



Microservice

Microservisní architektura je o redukci komplexity pomocí dekompozice systému na moduly, které jsou dobře ohraničené z hlediska fungování a účelu.

V business termínech by microservice měla realizovat jednu **business capability** (*schopnost či dovednost*)

Microservice nemá nic společného s velikostí, ale je o maximálním respektování **Single Responsibility Principle (SOLID)**

Microservisy fungují zcela samostatně a autonomě.

Microservice preferuje, aby měla vlastní databázi do které napřímo nepřistupuje žádný další systém (či microservice)

Microservice

Dvě microservisy mohou sdílet data a odpovědnost. Děje se tak přes třetí subjekt. Závislost je však co nejmenší a nejvolnější.

Dvě autonomní microservisy a sdílená databáze =>

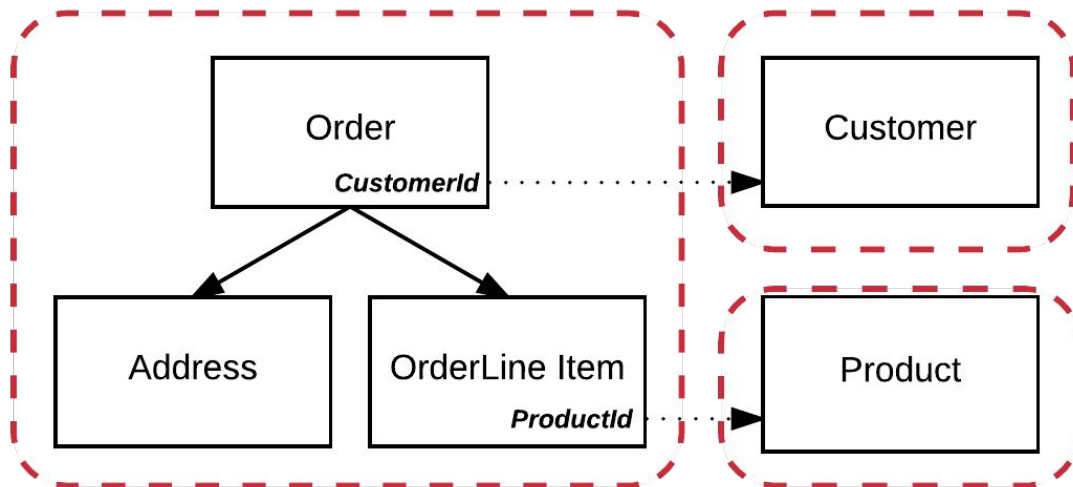


Microservice - Agregát

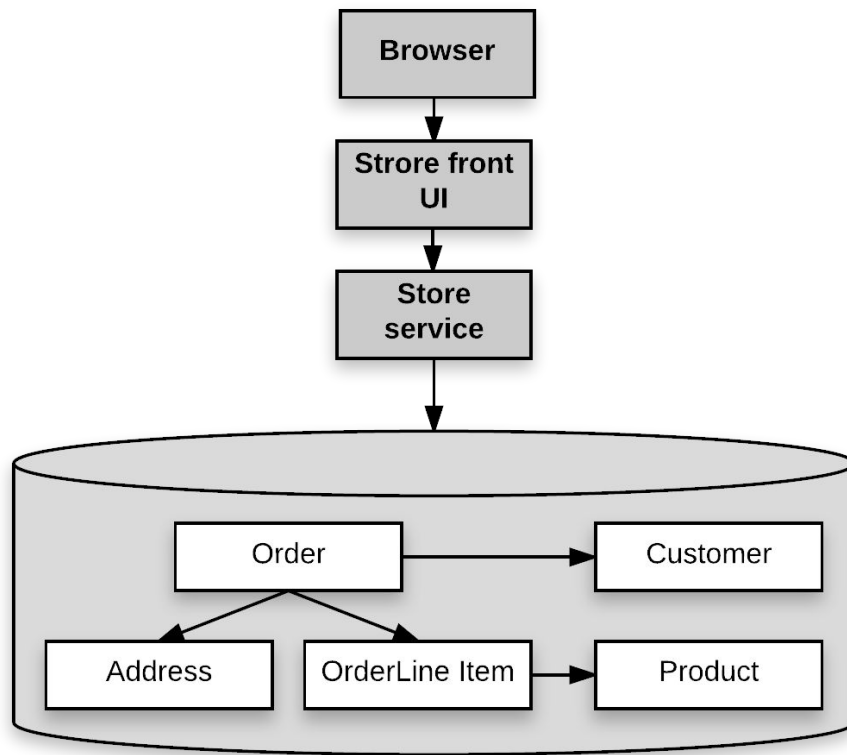
Agregát je graf objektů (nebo tabulek), ke kterým se mohou chovat jako k jednotce

Pravidla:

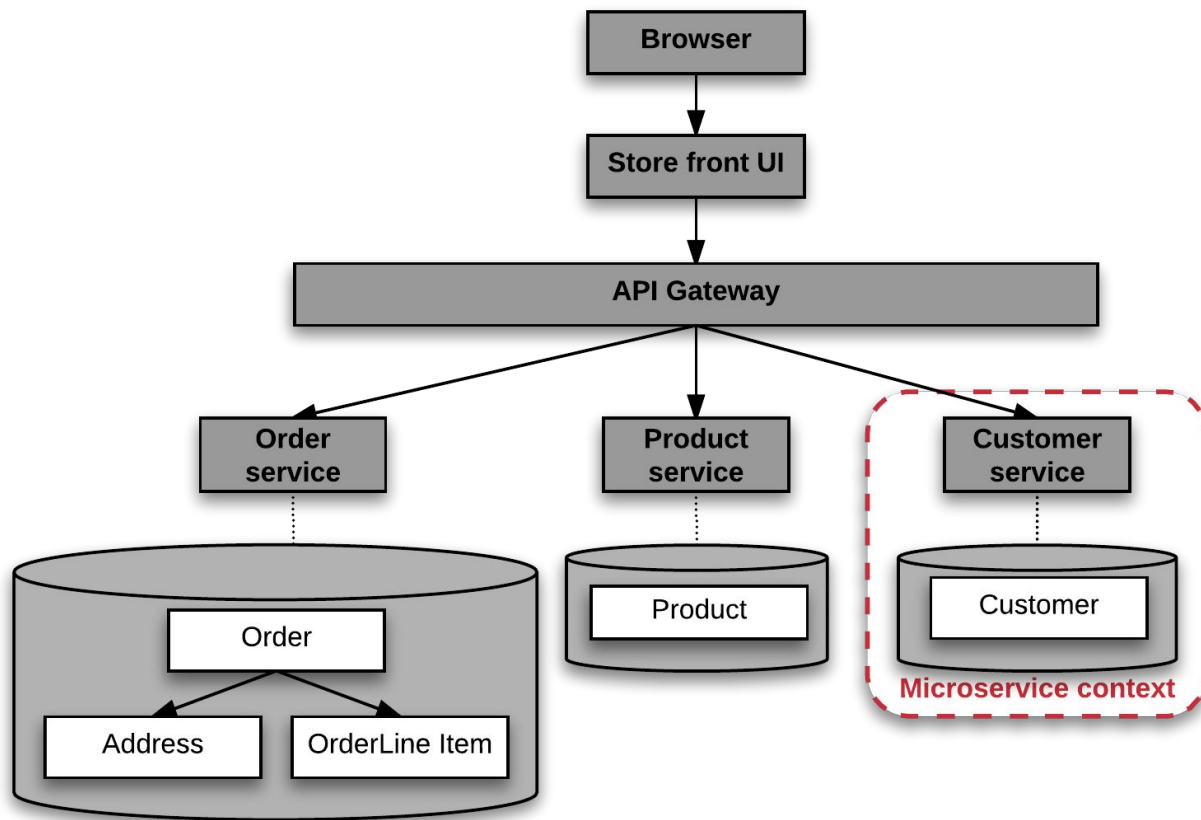
- Agregát má vždy jeden root (kořenový) objekt a sadu dalších navázaných objektů
- Agregát se linkuje na další agregáty pomocí **Id** jejich root objektů (*vzpoměňte na lazy loading pattern pomocí Ghost*)
- Jeden agregát = jeden command, který s ním pracuje
- Scope transakce = agregát



Klasický monolitický systém



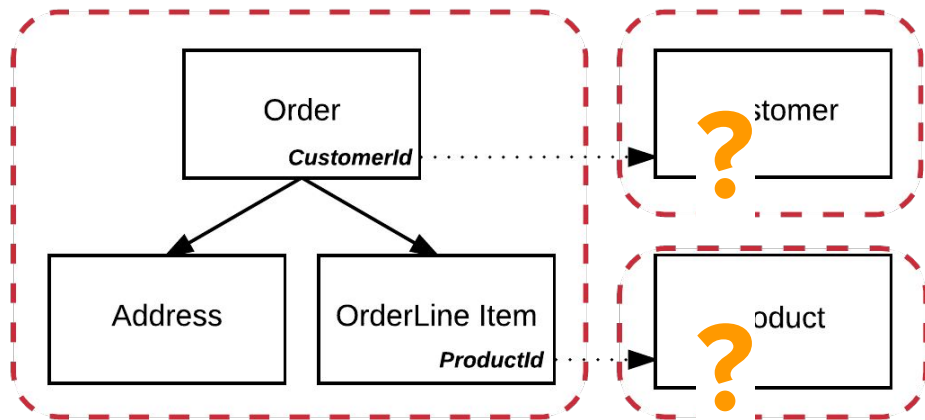
Dekompozice toho samého problému na microservices



Microservice

V klasickém monolitickém systému jsme mohli najednou měnit Order, Customer i Product, protože byly ve stejné databázi a přistupovali jsme k nim přes stejnou vrstvu.

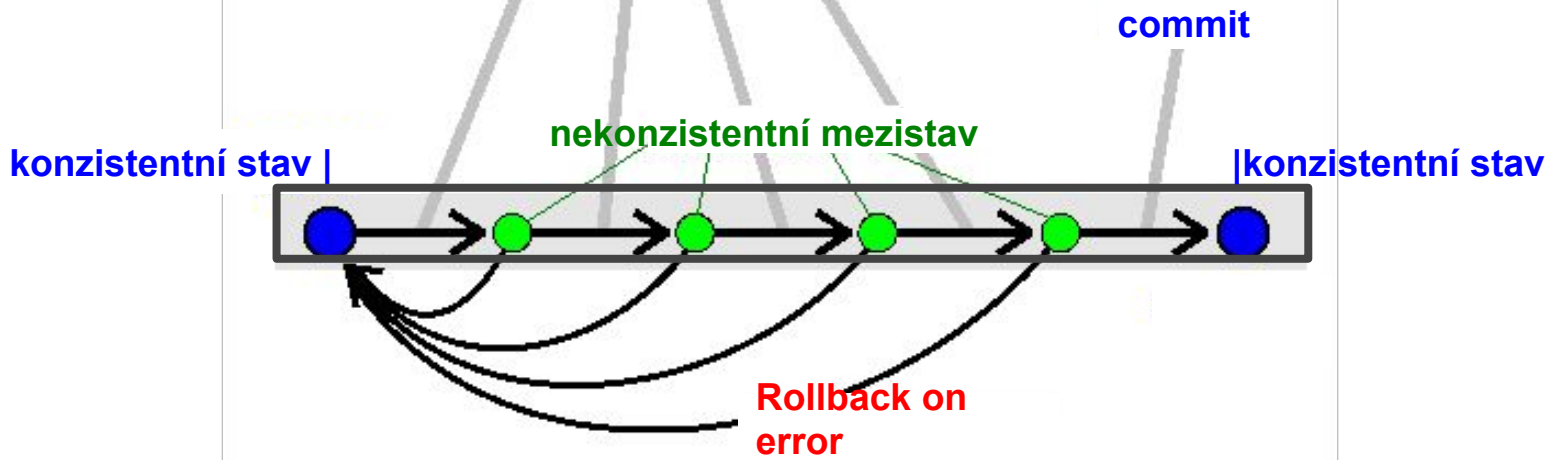
Jak zařídíme konzistentnost v systému dekomponovaném na microservisy?



Nejdříve je nutné pochopit co je transakce ve smyslu klasické relační databáze ...

Relační databáze a ACID

Transakce = sekvence operací (insert, update, delete...), které tvoří logický celek



ACID

Atomicity - změny prováděné v transakci jsou buď realizovány jako celek nebo vůbec

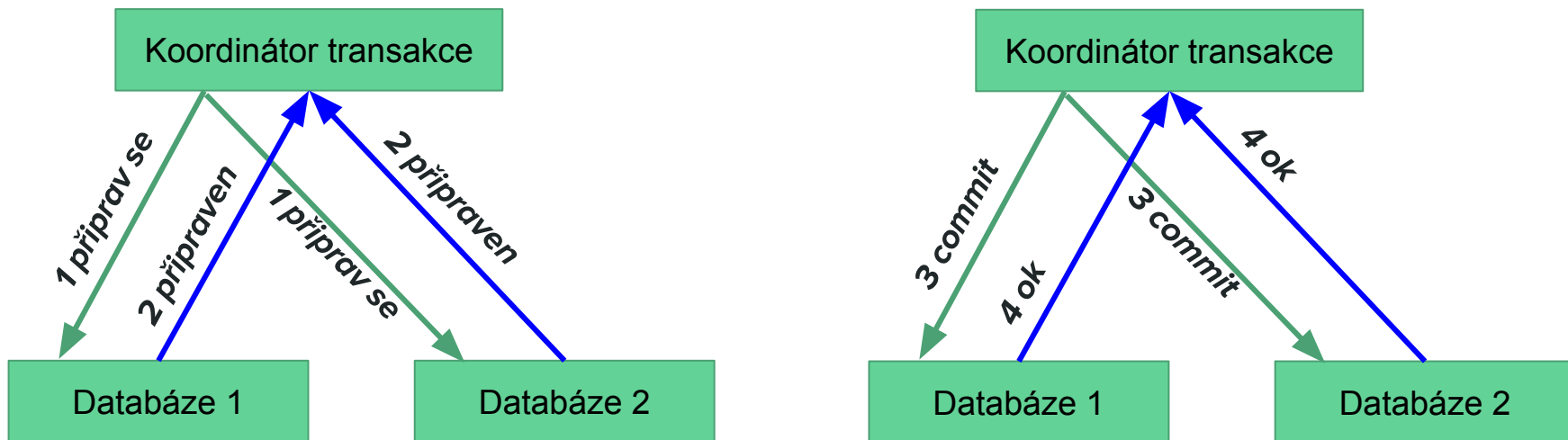
Consistency - databáze je konzistentní před a po transakci

Isolation - během transakce jsou data se kterými pracuje izolovány od dalších transakcí

Durability - úpravy provedené transakcí jsou natrvalo uloženy v databázi

Relační databáze a dvoufázový commit (2PC)

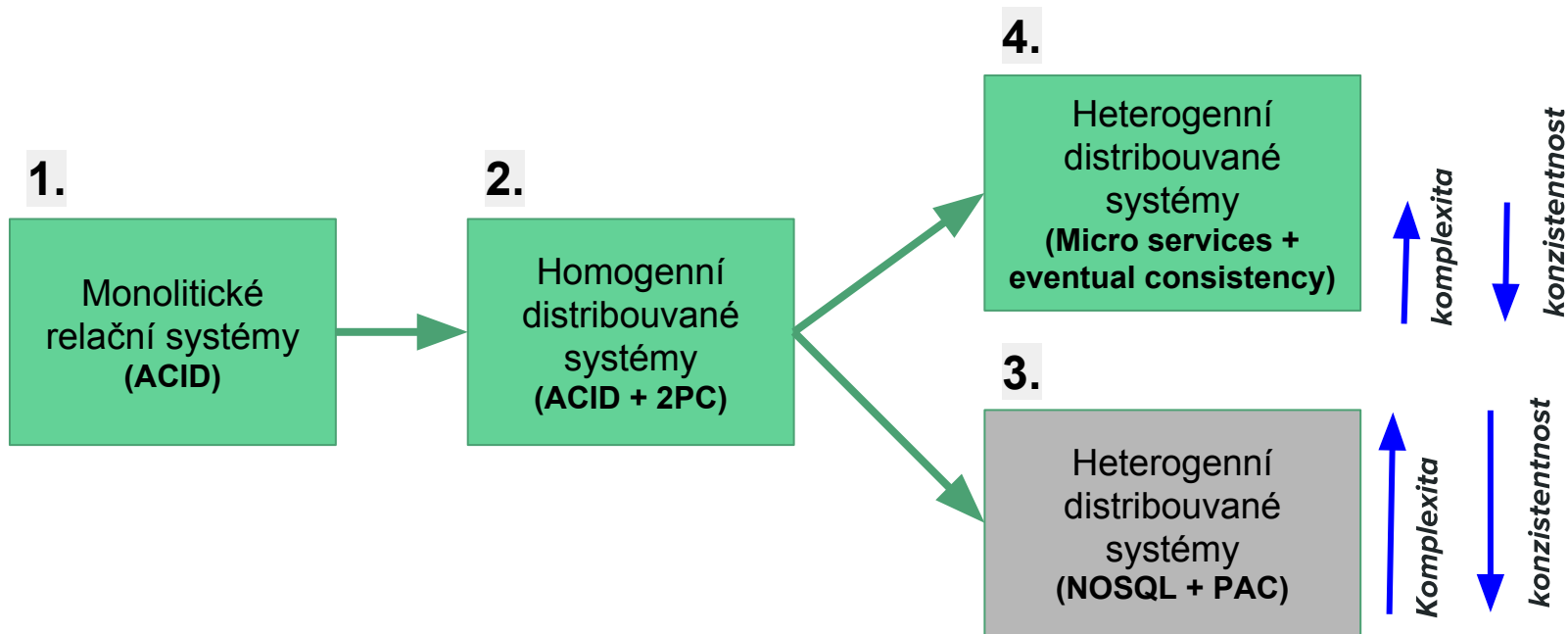
V případě, že všichni účastníci distribuované transakce potvrdí připravenost realizovat operace, tak se provádí *commit* na všech těchto systémech, jinak se neprovede na žádném



Proč 2PC není vždy použitelný:

- Existující aplikace většinou nepodporují transakce a dvoufázový commit - není to tzv. XA resource
- Koordinátor transakce zavádí *single point of failure* (při jeho výpadku nejde zapsat nikam)
- Protokol pro realizaci 2PC je velmi "upovídaný"
- Omezená propustnost, protože kvůli konzistentnosti musíme zamykat tabulky a serializovat operace

Geneze architektury



Sága a eventual consistency

Dočasně nekonzistentní systémy = **Eventual consistency**

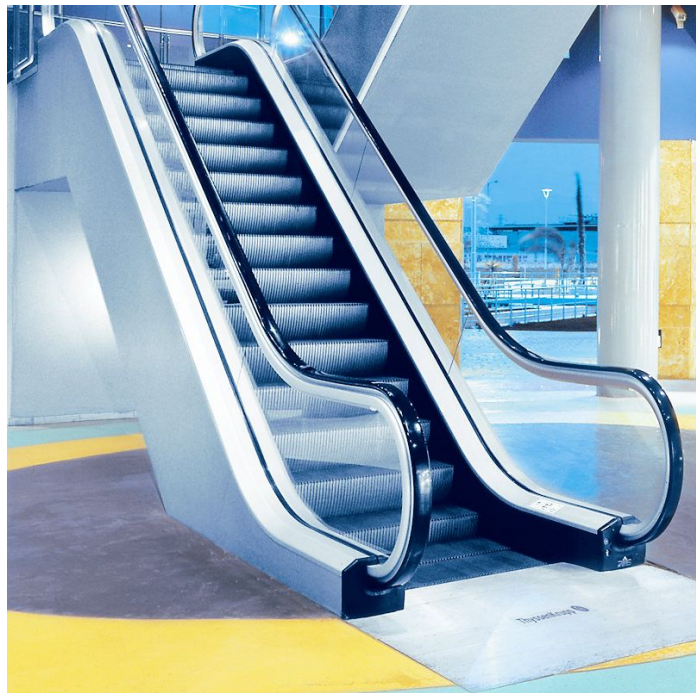
Z **technických** nebo **business důvodů** nejsme schopni po celou dobu zaručit konzistentnost

=> navrhne systém a business proces tak, aby byl použitelný i v nekonzistentních stavech (Inconsistent by design)

*Je použitelné i když není úplně funkční -
tady je to spíš náhoda než záměr ;-)* =>

Systém je konzistentní jen v některých stavech, kterými prochází. Jsou i systémy, které se do plně konzistentního stavu nedostanou nikdy

Např. hypotéka je konzistentní těsně před schválením. V tu chvíli ke všem nemovitostem existuje výpis z katastru, ke všem osobám na hypotéce existuje doklad o příjmech atd. Předtím byla hypotéka uložena, ale např. k některým osobám nebyl uložen příjem.



Sága a eventual consistency

Sága řeší konsensus mezi microservisami bez transakcí a to pomocí správné dekompozice business procesu a eventual consistency.

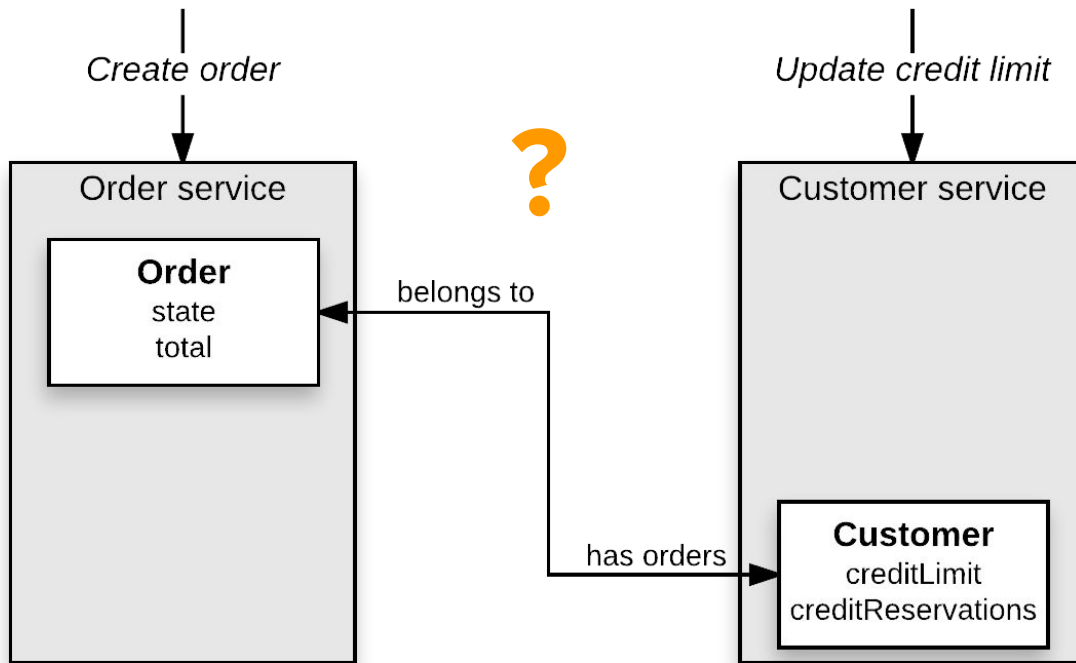
Sága rozdělí complexní business proces na menší akce a protiakce (counter actions), které koordinuje a řídí pomocí zpráv a timeoutů.

Business process => Saga, která obsahuje actions, counteractions, messages, timeouts

Operace v každém kroku business procesu má definovanou kompenzační operaci

Sága a eventual consistency

Zakládám objednávky a potřebuji zajistit, že suma otevřených objednávek není vyšší než kreditní limit zákazníka

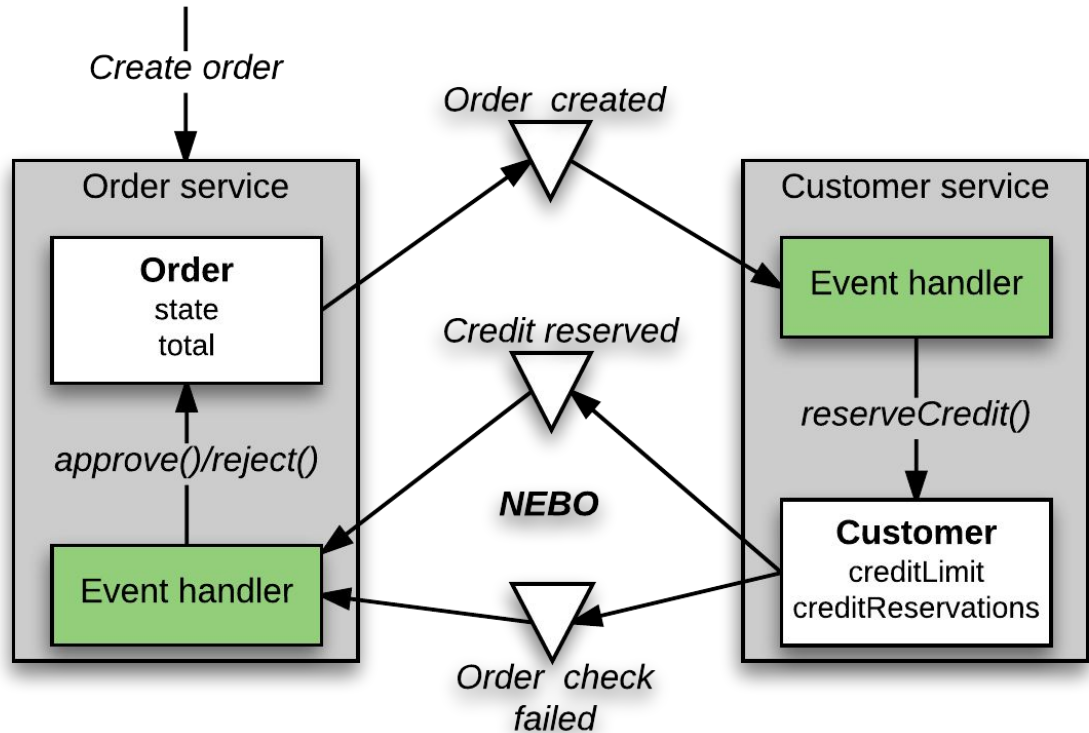


Sága a eventual consistency

Po založení objednávky je systém po nějakou dobu částečně nekonzistentní - mohl jsem eventuálně založit objednávku, která překračuje kreditní limit zákazníka.

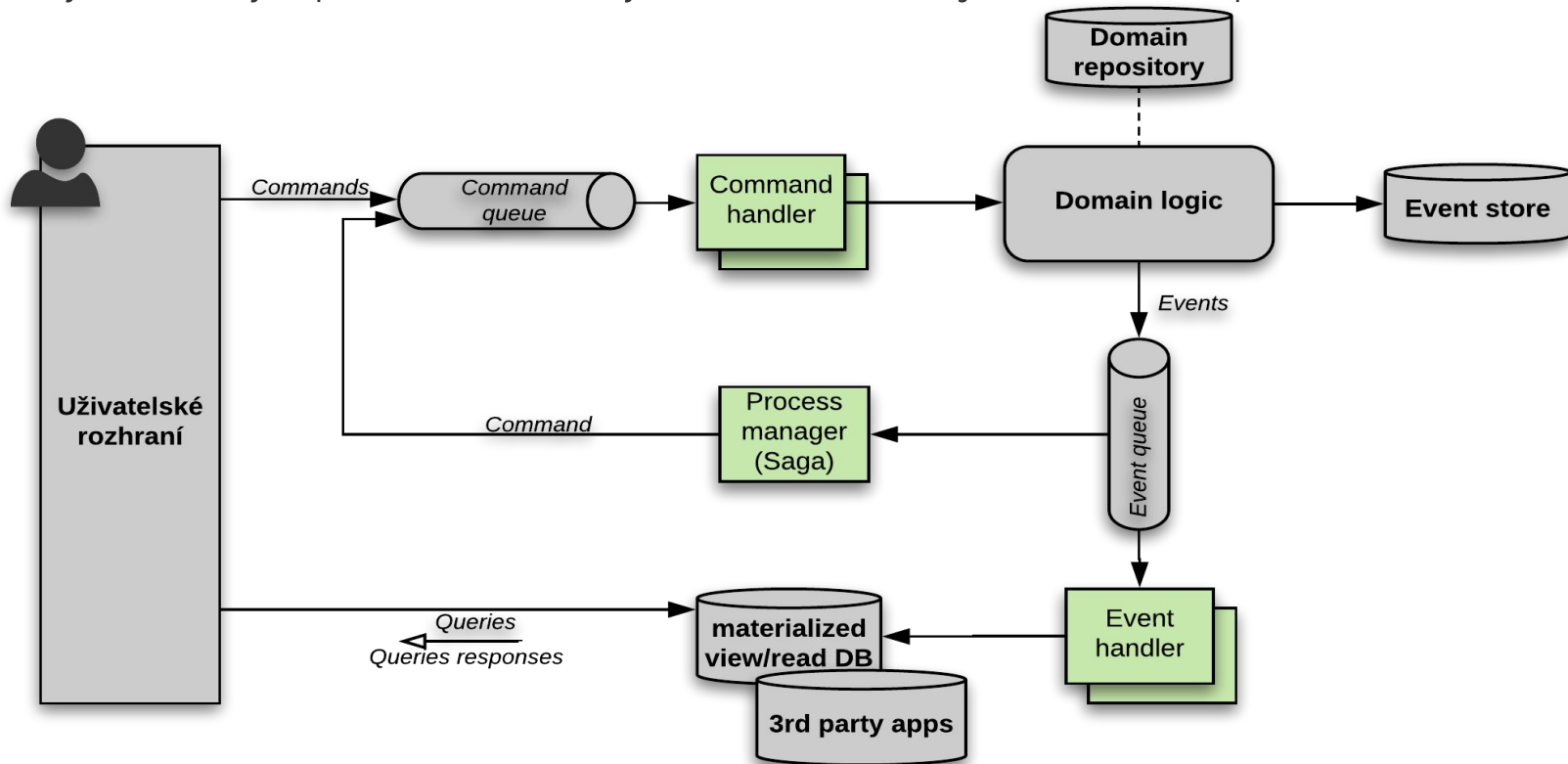
Toto řešení je možné, jelikož jsem **upravil business process a částečně degradoval user experience =>** Zákazník čeká na potvrzení objednávky.

Obecně platí, že od jistého stavu systému (parettovskly optimální řešení) nemůžu zlepšit některé jeho vlastnosti aniž bych degradoval jiné.



Materialized view + CQRS + Event sourcing + Saga

Po spojení výše uvedených patternů dohromady dostáváme následující architekturu aplikace



Materialized view + CQRS + Event sourcing + Saga

Command Queue - v této frontě se skladují commandy (pokyny co se má provést) 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 doménovou logikou, které provádí kód pro danou doménu (např. Logika pro zpracování klienta banky, logika pro spotřební úvěr atd.). 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 a ty jsou dále zpracovávány

Materialized view - datové repository optimalizované pro čtení např. z UI

Process Manager/Saga - implementace ságy, transformuje eventy do akcí nad doménovou logikou