OMO

9 - Map/filter/reduce, CQRS

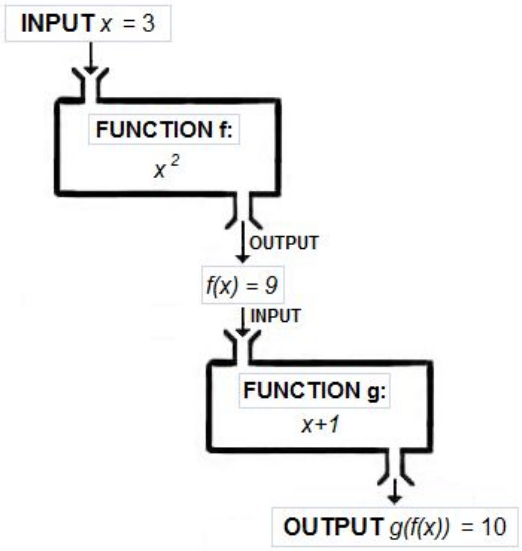
- Map/filter/reduce v java
- Materialized view
- Event sourcing
- CQRS

Ing. David Kadleček, PhD
kadlecd@fel.cvut.cz, david.kadlecek@cz.ibm.com

Map/filter/reduce v Java

Map/filter/reduce v Java = Java 1.8 streams API

Funkcionální přístup (řetězíme operace do sebe) **+** **Abstrakce kontrolního flow**



Cykly (for, while …) - procházíme a řešíme jak

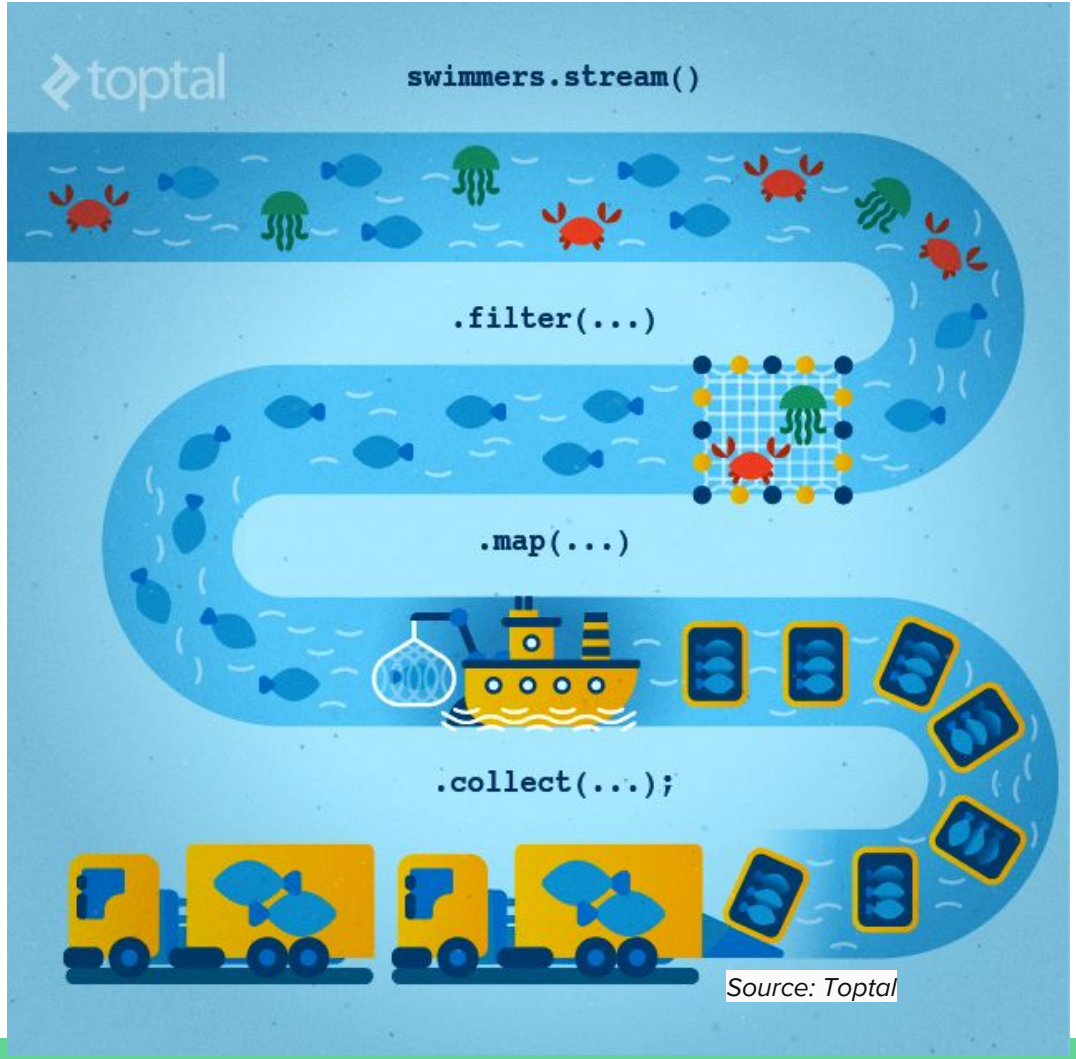
⬇

Iterátory - procházíme

⬇

Map/filter/reduce - nemusíme dělat nic

Map/filter/reduce v Java



Source: Toptal

Jak vytvořit stream

Stream lze získat z libovolné kolekce:

```
Collection<String> names = Arrays.asList("John", "David", "Martin");
Stream<String> streamingNames = names.stream();
```

Stream lze vytvořit z posloupnosti pevně daných prvků:

```
Stream<String> streamingNames = Stream.of("John", "David", "Martin")
```

Další možností definice streamu je ***indukce***. Stream je zadán prvním prvkem a unární operací, která je aplikována na poslední vytvořený prvek za účelem vytvoření prvku následujícího.

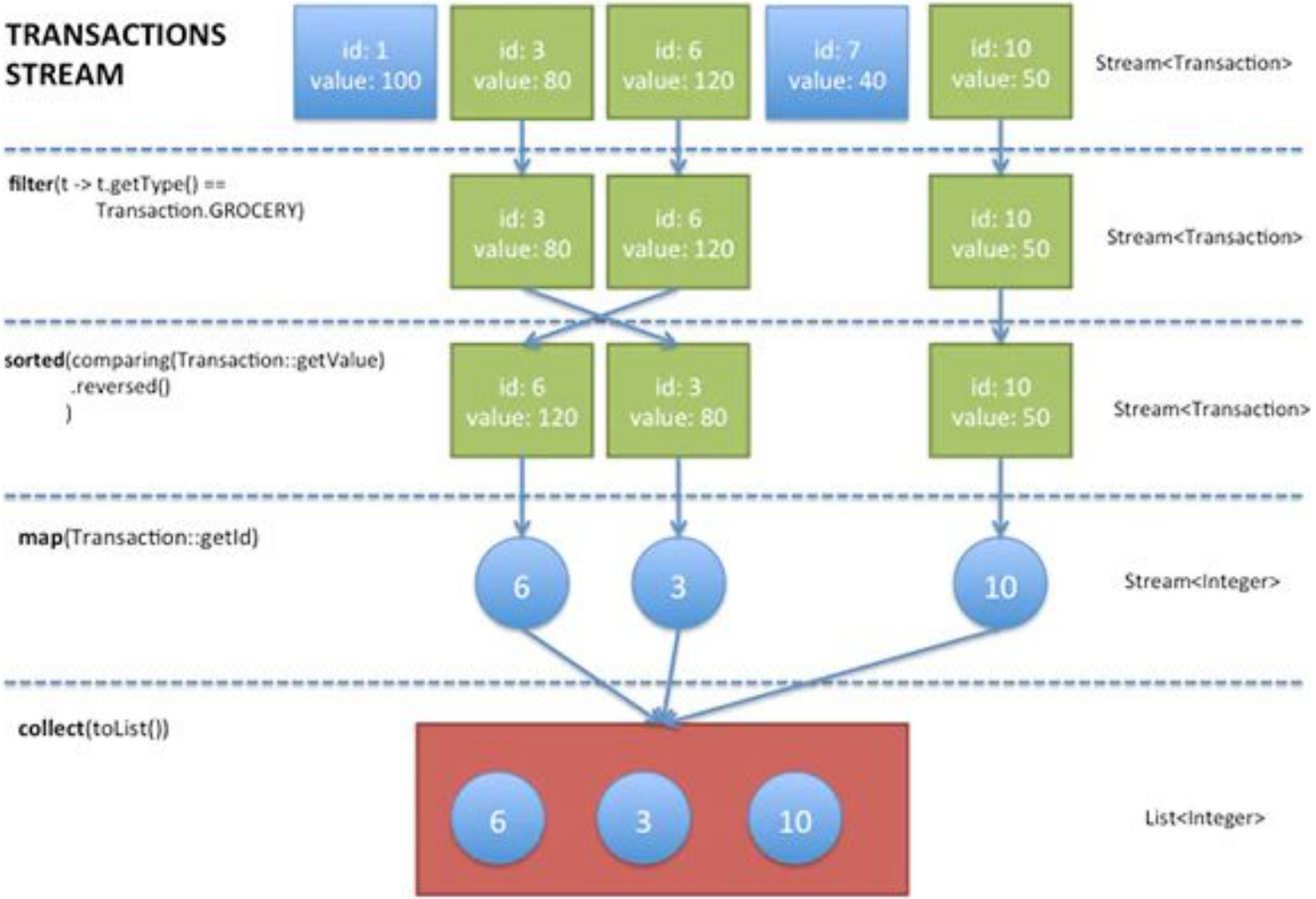
```
Stream<Boolean> streamingBooleans = Stream.iterate(false, i -> !i);
```

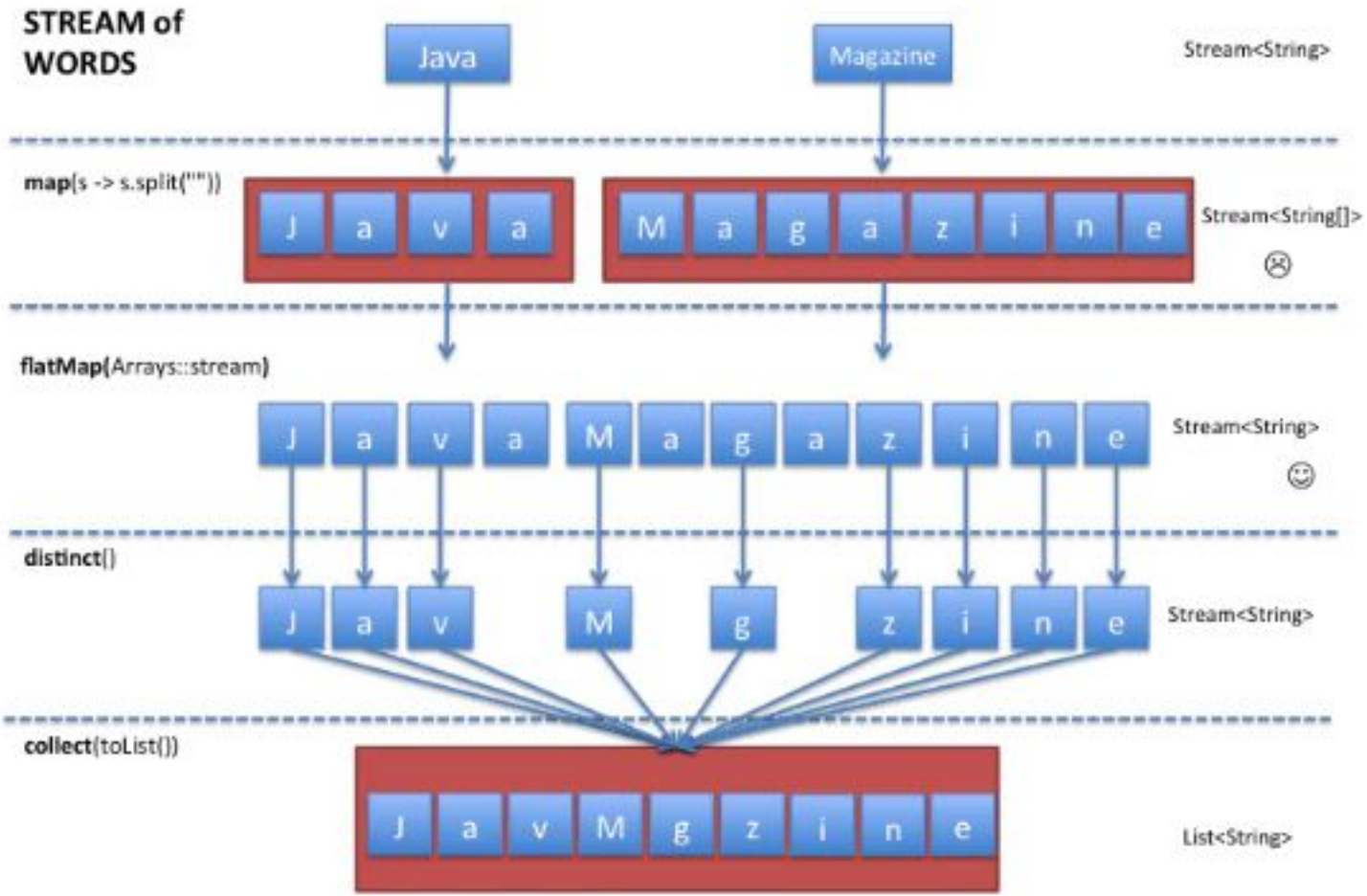
Příklad map/filter/reduce

Předpokládejme, že existuje třída Person s atributy name (jméno), age (věk), city (město). Dále mějme kolekci nějakých lidí uloženou v kolekci people.

Chceme najít jméno nejstaršího člověka z Washingtonu:

```
String name = people
        // převést na stream
        .stream()
        // dál propustit pouze lidi z Washingtonu
        .filter(p -> p.getCity().equals("Washington"))
        // seřadit podle věku sestupně
        .sorted(Comparator.comparingInt((Person p) -> p.getAge()).reversed())
        // najít první takovou osobu
        .findFirst()
        .get()
        // získat jméno osoby
        .getName();
```


TRANSACTIONS STREAM
id: 1 value: 100
id: 3 value: 80
id: 6 value: 120
id: 7 value: 40
id: 10 value: 50
Stream<Transaction>
filter(t -> t.getType() == Transaction.GROCERY)
id: 3 value: 80
id: 6 value: 120
id: 10 value: 50
Stream<Transaction>
sorted(comparing(Transaction::getValue) .reversed())
id: 6 value: 120
id: 3 value: 80
id: 10 value: 50
Stream<Transaction>
map(Transaction::getId)
6
3
10
Stream<Integer>
collect(toList())
6
3
10
List<Integer>

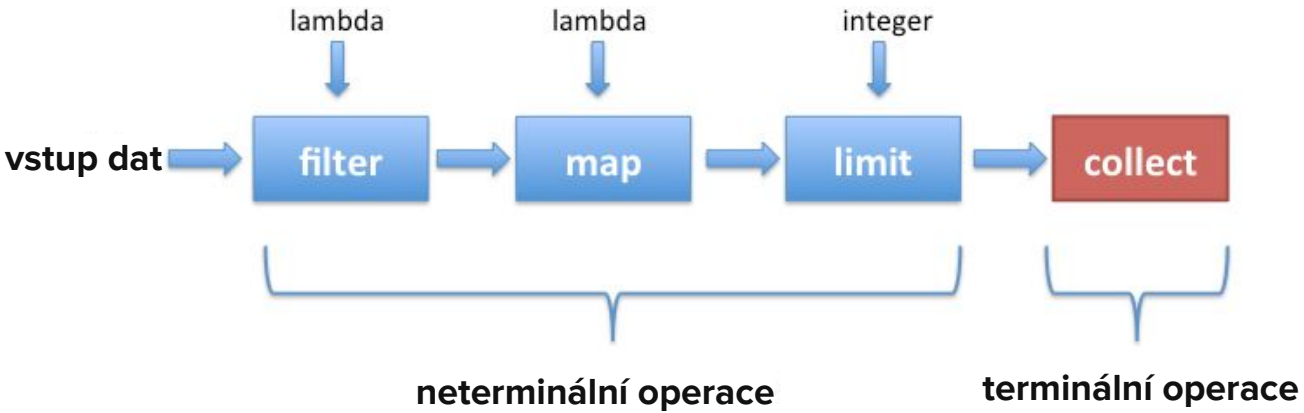

STREAM of WORDS
Java
Magazine
Stream<String>
map(s -> s.split(""))
J a v a
M a g a z i n e
Stream<String[]>
☹
flatMap(Arrays::stream)
J a v a M a g a z i n e
Stream<String>
☺
distinct()
J a v M g z i n e
Stream<String>
collect(toList())
J a v M g z i n e
List<String>

Pipeline

Použitím streamu vzniká tzv. **pipeline**, což je konečná posloupnost operací aplikovaná na nějaký stream.

Operace se dělí do skupin:

- **terminální operace** (terminal) - má postranní efekt nebo produkuje hodnotu; po spuštění této operace je proud uzavřen a nelze jej použít (např. forEach)
- **neterminální operace** (intermediate) - nemá žádné postranní efekty; vytváří vždy nový stream
  - stavové (stateful) - stav operace je ovliněn procházejícími prvky (např. sorted)
  - bezstavové (stateless) - operace nemá žádný vnitřní stav (např. filter)



Map/filter/reduce operace

Mějme abstraktní datový typ Seq<E> reprezentující *sekvenci* elementů typu E.

Např., [1, 2, 3, 4] ∈ Seq<Integer> .

Map aplikuje unární funkci na každý element sekvence a vrací novou sekvenci výsledků ve stejném pořadí:

map : (E ➜ F) × Seq<E> ➜ Seq<F>

Filtr testuje každý element unárním predikátem. Elementy, které vyhovují jsou ponechána a ostatní odstraněny. Vrácen je nový list.

filter : (E ➜ boolean) × Seq<E> ➜ Seq<E>

Map/filter/reduce operace

Reduce kombinuje elementy sekvence dohromady s pomocí binární operace. Dále bere v potaz inicializační hodnotu, kterou inicializuje redukci nebo vrátí zpět u prázdné sekvence.

reduce : (F × E ➡ F) × Seq<E> × F ➡ F

reduce(f, list, init) kombinuje elementy listu zleva doprava:

result 0 = init
result 1 = f(result 0 , list[0])
result 2 = f(result 1 , list[1])
...
result n = f(result n-1 , list[n-1])

Sečtení kolekce čísel:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
Integer sum = numbers.stream().reduce(0,(a,b)->a+b);
```

Příklady operací

Operace	Popis	Druh
filter	filtruje prvky podle zadaného predikátu	bezstavová-NT
map	převádí prvky na jiné pomocí zadaného zobrazení	bezstavová-NT
flatMap	Jako map, ale vnořené kolekce převede do jediné kolekce	bezstavová-NT
groupBy	Seskupení podle vybraného atributu	bezstavová-NT
reduce	generická operace redukce	terminální
max	konkrétní operace redukce	terminální
limit	omezí maximální délku streamu na zadaný počet prvků	terminální
forEach	všechny prvky streamu postupně odešle konzumentovi	terminální
sorted	prvky v streamu budou do dalších operací předávány seřazené	stavová-NT
findFirst	vezme první prvek z streamu	bezstavová-NT
count	spočítá prvky v streamu	bezstavová-NT

Další příklady map/filter/reduce

Průměrný věk

```
double avgAge = people
        // převést na proud
        .stream()
        // od osoby získat její věk
        .mapToInt(p -> p.getAge())
        // z čísel vypočítat průměr
        .average()
        .getAsDouble();
```

Setřídění

```
List<String>
myList=Arrays.asList("a1","a2","b1","c2","c1");
myList
    .stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

```
OUTPUT:
C1
C2
```

Kolektory

Kolektor je třída, která agreguje prvky ze streamu. Existují kolektory, které prvky jednoduše uloží do seznamu, provedou seskupení podle nějakého kritéria a tyto skupiny uloží do mapy, a podobně. Lze vytvářet i vlastní kolektory.

```
// sesbírá jména lidí do seznamu
List<String> list = people.stream().map(Person::getName).collect(Collectors.toList());
```

```
// seskupí osoby do mapy podle města, kde žijí
Map<String, List<Person>> personByCity = people
    .stream()
    .collect(Collectors.groupingBy(Person::getCity));
```

```
// do mapy uloží počet osob v každém městě
Map<String, Integer> peoplePerCity = people
    .stream()
    .collect(Collectors.groupingBy(Person::getCity, Collectors.counting()));
```

Performance pohled

Stream může být nastaven jako **paralelní**

```
Arrays.asList("John", "David", "Martin").parallelStream()
```

Neterminální operace (intermediate) jsou **lazy**

```
//Created a list of students
Stream<String> streamOfNames = students.stream()
                .map(student -> {
                    System.out.println("In Map - " + student.getName());
                    return student.getName();
                });
//Just to add some delay
for (int i = 1; i <= 5; i++) {
   Thread.sleep(1000);
   System.out.println(i + " sec");
}
//Called a terminal operation on the stream
streamOfNames.collect(Collectors.toList());
```

Operace se na streamu začnou provádět až ve chvíli, kdy potřebujeme výstup

=>

```
OUTPUT:
1 sec
2 sec
3 sec
4 sec
5 sec
In Map - Tom
In Map - Chris
In Map - Dave
```

```
List<String> ids = students.stream()
        .filter(s -> {System.out.println("filter - "+s); return s.getAge() > 20;})
        .map(s -> {System.out.println("map - "+s); return s.getName();})
        .limit(3)
        .collect(Collectors.toList());
```

Prvky pokud možno proplouvají pipelinou po jedné od vstupu až na výstup. *Id - 8* prvního studenta prochází filtrem a okamžitě pokračuje do mapy. Pak *id - 9*, pak *id - 10* (neprošlo filtrem) atd.

=>

```
OUTPUT:
filter - 8
map - 8
filter - 9
map - 9
filter - 10
filter - 11
map - 11
```

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

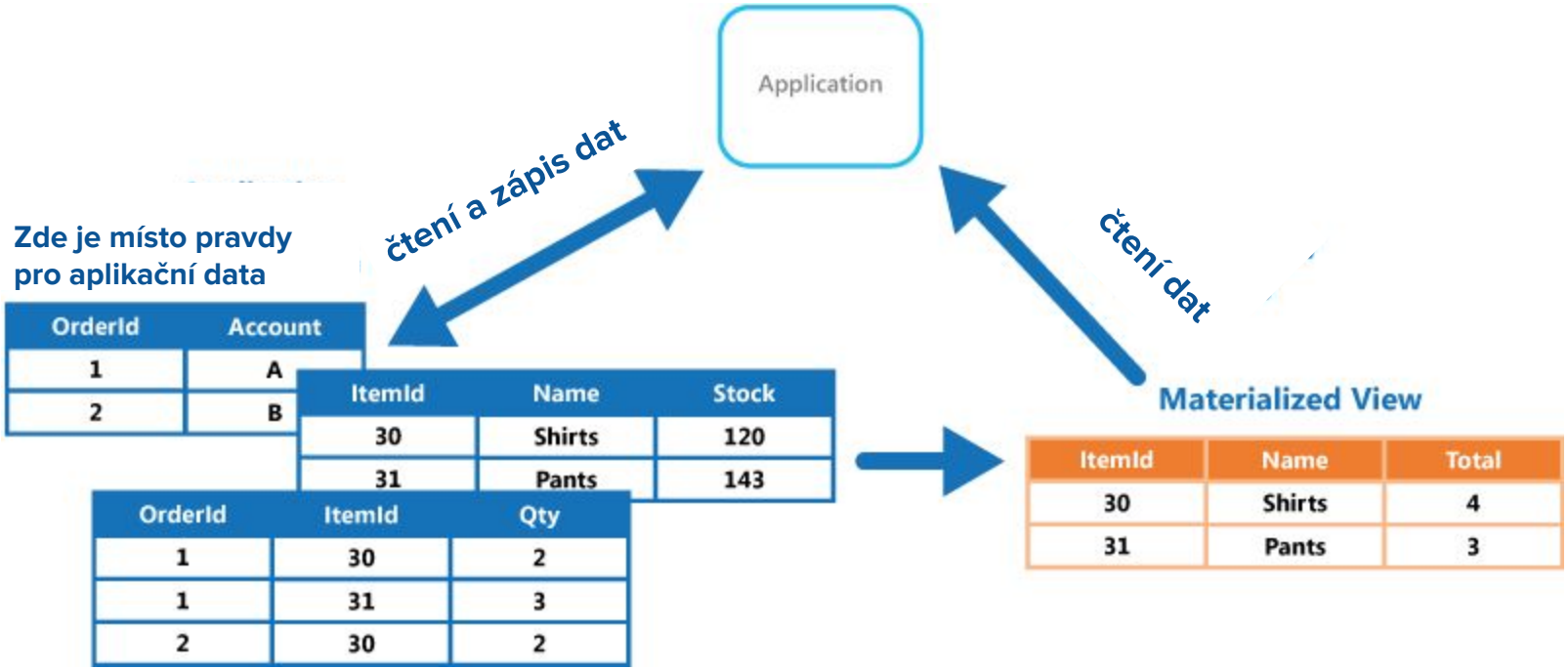
=> Z místa pravdy dat si dopředu vytvoříme pohled na data, který obsahuje pouze data které potřebujeme.

=> Data jsou v modelu, který se nám dobře čte.

=> Tento pohled používáme pouze na čtení.

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

Materializovaný pohled (Materialized view)



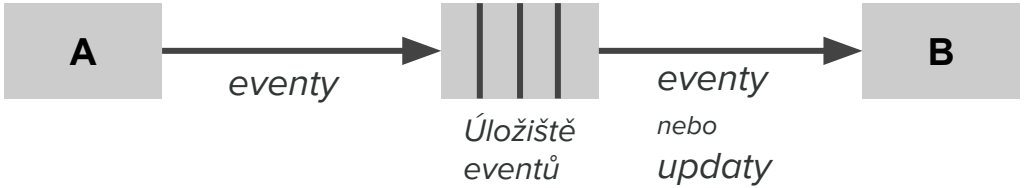
Event sourcing

Klasický přístup při změně dat:

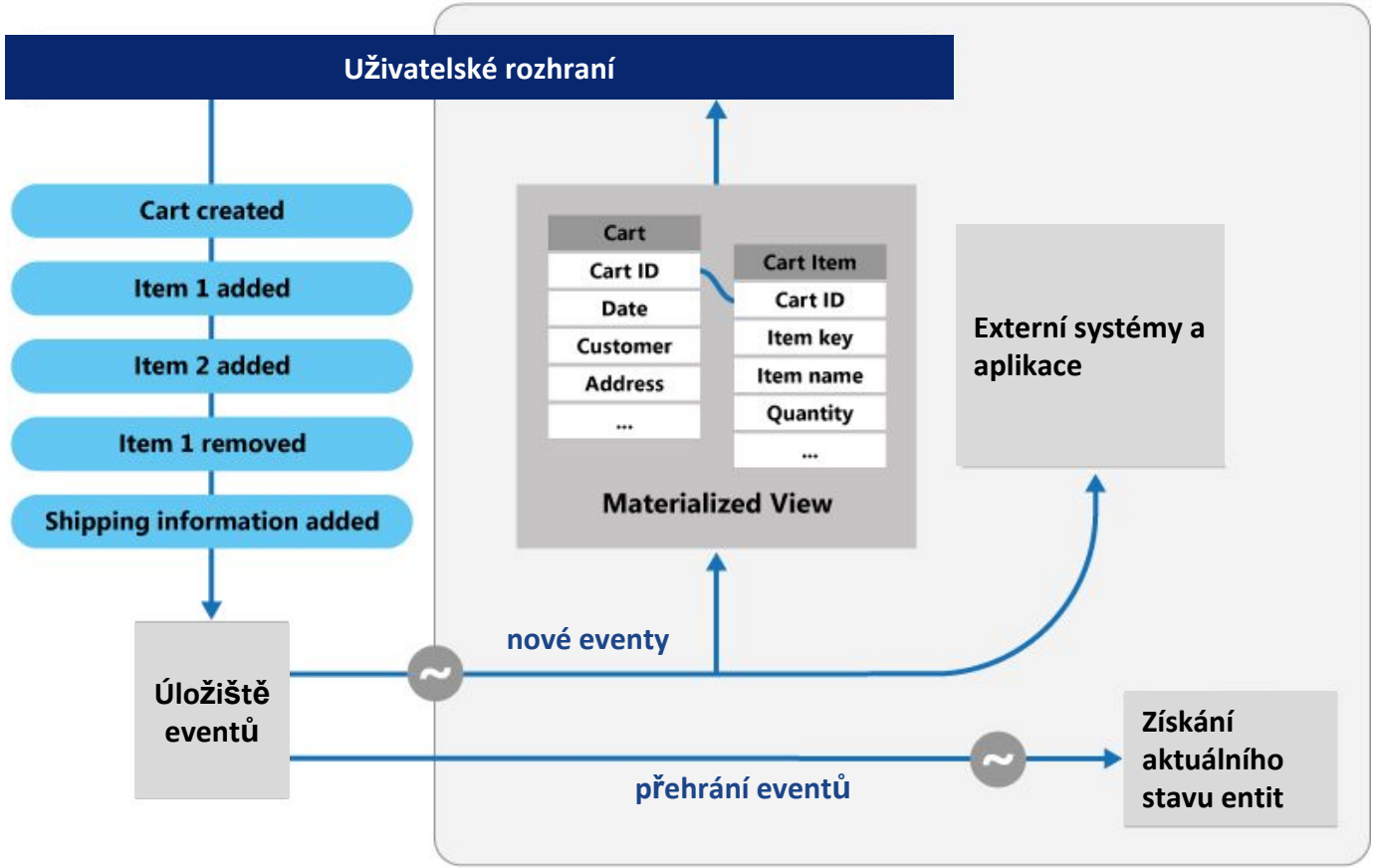
- první aplikace přímo updatuje stav druhé aplikace

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 ...)

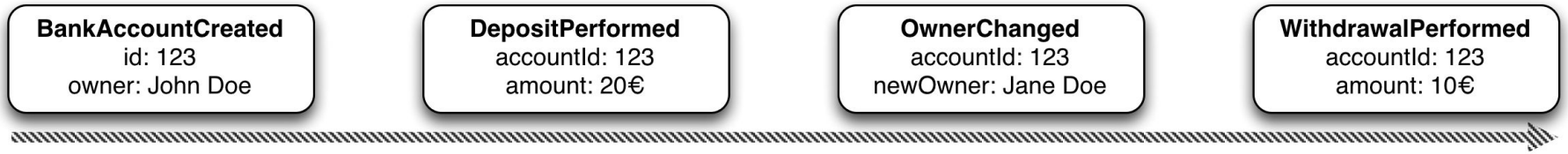


Event sourcing pattern

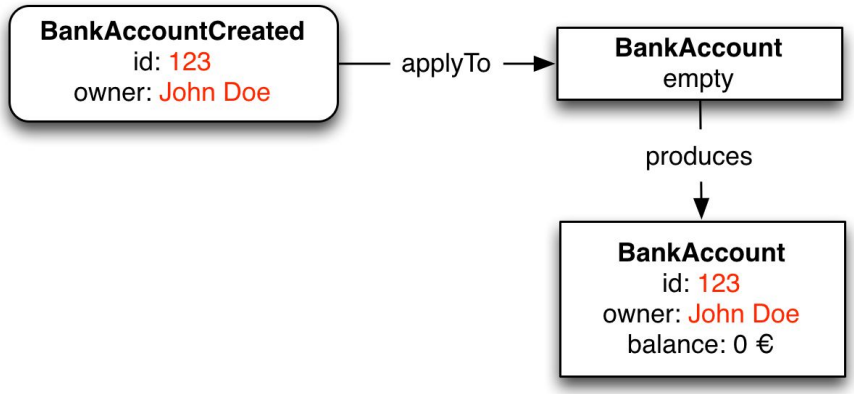


Event sourcing

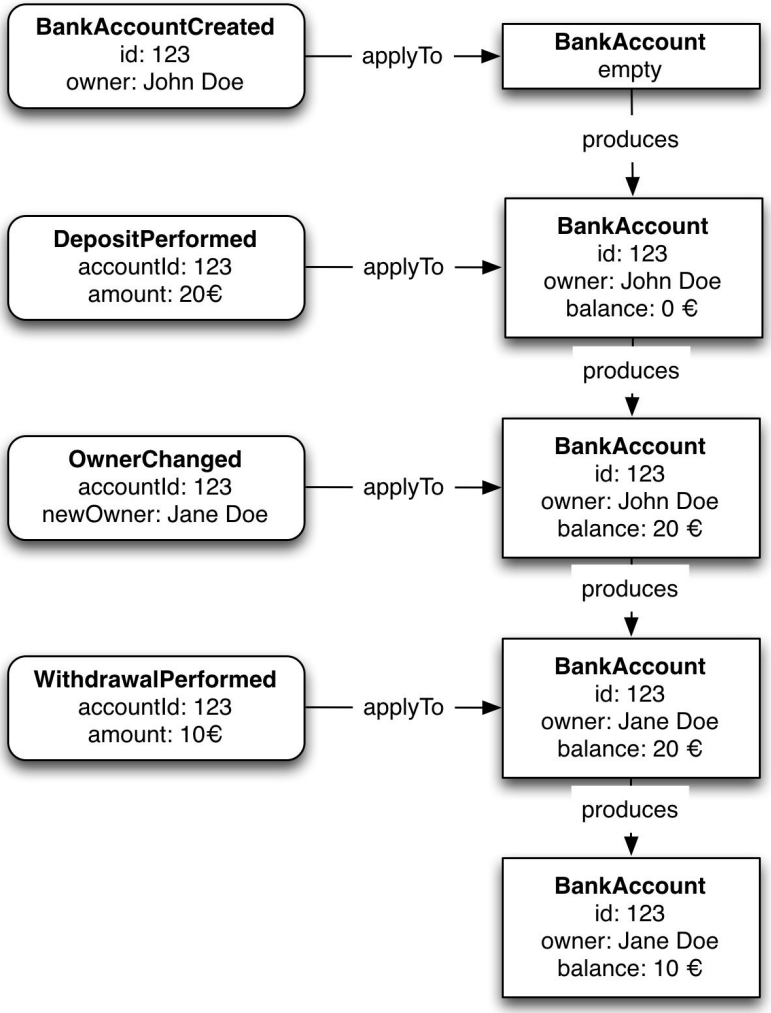
Sekvence eventů >>>



Aplikace eventu



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. abych neposílal klientovi 2x ten samý email => řešením je např. oddělení side efektů a stavových změn
- Úložiště eventů se ***nehodí na generování reportů a dotazování na data*** => vytvoříme si z eventů čtecí úložiště

CQRS

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

Hlavní idea CQRS je:

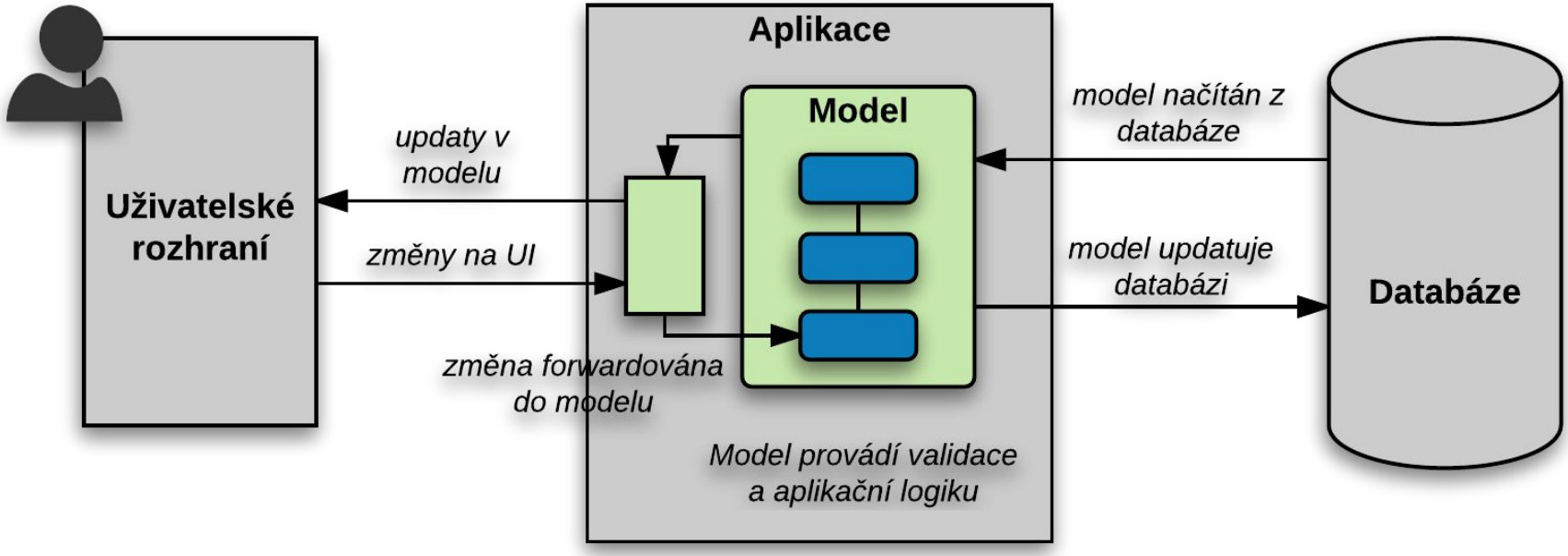
Metoda by měla buď změnit stav objektu nebo vrátit výsledek - ale ne obojí najednou - odpovídání na otázku nemění otázku. Když prodeme takovouto segregaci, tak budeme mít vždy dva typy metod

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

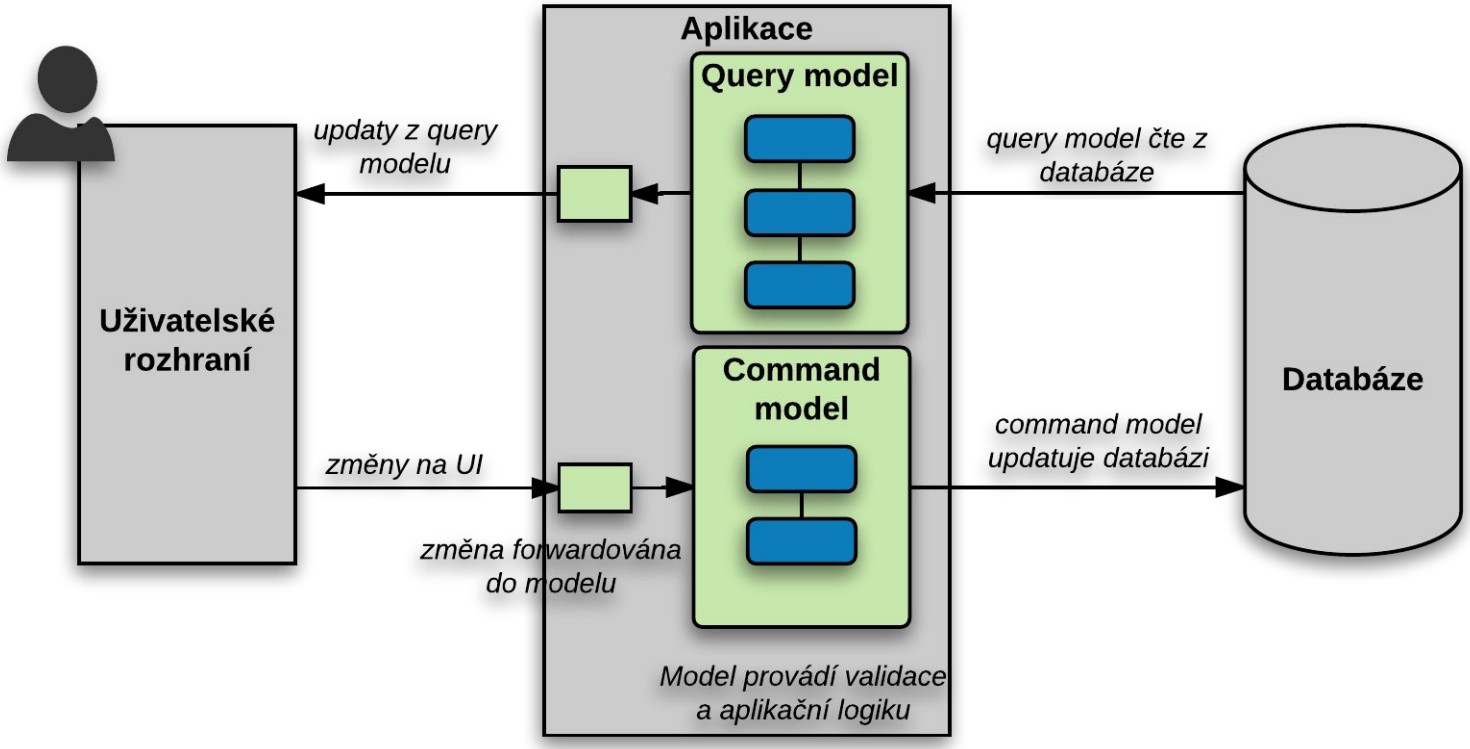
Důvody pro CQRS:

- 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 ubírají si výkon

CQRS


Aplikace
Model
Uživatelské rozhraní
updaty v modelu
změny na UI
změna forwardována do modelu
model načítán z databáze
model updatuje databázi
Databáze
Model provádí validace a aplikační logiku

CQRS


Aplikace
Query model
Command model
Uživatelské rozhraní
Databáze
updaty z query modelu
query model čte z databáze
změny na UI
command model updatuje databázi
změna forwardována do modelu
Model provádí validace a aplikační logiku